Элайджи Микс, Энн-Мари Дюфур

D3.js в действии

D3.js in Action

THIRD EDITION

ELIJAH MEEKS
ANNE-MARIE DUFOUR
Foreword by ANDY KIRK



MANNING
Shelter Island

D3.js в действии

ЭЛАЙДЖИ МИКС
ЭНН-МАРИ ДЮФУР
Предисловие Энди Кирка

ДМК Пресс
издательство
Москва, 2025

УДК 004.738:004.92D3.js
ББК 32.973.4
М59

Микс Э., Дюфур Э.-М.
М59 D3.js в действии / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2025. – 726 с.: ил.

ISBN 978-5-93700-225-9

Книга рассказывает о библиотеке D3.js, которая позволяет создавать сложную инфографику, диаграммы и визуализации данных с применением стандартных инструментов разработки веб-интерфейсов, включая JavaScript, HTML и CSS. Вы научитесь воспроизводить анимационные эффекты и настраивать адаптивные отображения на мобильных устройствах. Каждая глава содержит законченный проект визуализации, иллюстрирующий применение новых навыков на практике.

Издание адресовано веб-разработчикам, владеющим навыками использования HTML, CSS и JavaScript.

УДК 004.738:004.92D3.js
ББК 32.973.4

Copyright © DMK Press 2025. Authorized translation of the English edition.

© 2024 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-6334-3917-7 (англ.)
ISBN 978-5-93700-225-9 (рус.)

© Manning Publications, 2024
© Перевод, оформление, издание, ДМК Пресс, 2025

Моим трем началам

Оглавление

https://t.me/it_boooks/2

Часть I ОСНОВЫ D3.js 24

1 ▪ Введение в D3.js 25

2 ▪ Манипулирование деревом DOM 73

3 ▪ Работа с данными 100

4 ▪ Рисование линий и дуг 151

5 ▪ Круговые и стековые макеты 194

6 ▪ Визуализация распределений 239

Часть II НОВЫЕ СТАНДАРТЫ 279

7 ▪ Интерактивные визуализации 280

8 ▪ Интеграция D3 с фреймворками пользовательских интерфейсов 329

9 ▪ Адаптивные визуализации 375

10 ▪ Доступность визуализаций для людей с ограниченными возможностями 405

Часть III СЛОЖНЫЕ ВИЗУАЛИЗАЦИИ ДАННЫХ 452

11 ▪ Иерархические визуализации 453

12 ▪ Сетевые визуализации 493

13 ▪ Визуализации геопространственной информации 530

Часть IV ПРОДВИНУТЫЕ МЕТОДЫ ВИЗУАЛИЗАЦИИ 577

14 ▪ Создание нестандартных визуализаций 578

15 ▪ Отображение визуализаций с помощью Canvas 628

Содержание

Предисловие ... 15
Вступление ... 17
Благодарности ... 18
О книге ... 20
Об авторах ... 22
Об иллюстрации на обложке ... 23

Часть I ОСНОВЫ D3.js ... 24

1 ***Введение в D3.js*** ... 25
1.1 Что такое D3.js? ... 27
1.1.1 Необходимость в визуализации данных для веба ... 27
1.1.2 Когда следует использовать D3.js? ... 29
1.1.3 Как работает D3.js ... 31
1.2 Экосистема D3: что нужно знать для начала работы ... 36
1.2.1 HTML и DOM ... 37
1.2.2 Масштабируемая векторная графика ... 38
1.2.3 Canvas и WebGL ... 57
1.2.4 CSS ... 57
1.2.5 JavaScript ... 58
1.2.6 Фреймворки Node и JavaScript ... 63
1.2.7 Блокноты Observable ... 64
1.3 Передовые практики визуализации данных ... 64
Итоги ... 72

2 ***Манипулирование деревом DOM*** ... 73
2.1 Наша первая визуализация D3 ... 74
2.2 Подготовка среды ... 75
2.2.1 Структура нашего первого проекта D3 ... 77

2.2.2 Загрузка D3 в проект ... 78
2.3 Выбор элементов ... 80
2.4 Добавление элементов в выборку ... 84
2.5 Установка и изменение атрибутов ... 86
2.6 Установка и изменение стилей ... 91
Итоги ... 98

3 *Работа с данными* ... 100
3.1 Изучение данных ... 102
3.1.1 Поиск данных ... 102
3.1.2 Типы данных ... 102
3.1.3 Форматы и структуры данных ... 104
3.2 Подготовка данных ... 108
3.2.1 Загрузка набора данных в проект D3 ... 109
3.2.2 Форматирование набора данных ... 111
3.2.3 Оценка набора данных ... 113
3.3 Привязка данных к элементам DOM ... 117
3.3.1 Динамическая настройка атрибутов DOM с помощью данных ... 121
3.4 Адаптация данных под размеры экрана ... 126
3.4.1 Шкалы ... 127
3.4.2 Линейная шкала ... 130
3.4.3 Интервальная шкала ... 134
3.5 Добавление меток на диаграмму ... 137
Итоги ... 149

4 *Рисование линий и дуг* ... 151
4.1 Добавление осей ... 152
4.1.1 Соглашение о границах и отступах ... 155
4.1.2 Генерирование осей ... 158
4.2 Построение линейной диаграммы ... 166
4.2.1 Использование генератора линий ... 168
4.2.2 Интерполяция точек данных в кривую ... 170
4.3 Рисование области размаха ... 173
4.3.1 Использование генератора области размаха ... 173
4.3.2 Улучшение читаемости с помощью меток ... 176
4.4 Рисование дуг ... 179
4.4.1 Полярная система координат ... 180
4.4.2 Использование генератора дуг ... 183
4.4.3 Вычисление центра дуги ... 185
Итоги ... 192

5 *Круговые и стековые макеты* ... 194
5.1 Создание круговых и кольцевых диаграмм ... 197
5.1.1 Подготовительные шаги ... 197
5.1.2 Генератор кругового макета ... 200
5.1.3 Рисование дуг ... 205
5.1.4 Добавление меток ... 207

5.2 Создание стековых диаграмм ... 210
5.2.1 Генератор стекового макета ... 211
5.2.2 Рисование составной столбчатой диаграммы ... 215
5.2.3 Рисование потокового графика ... 218
5.2.4 Функции доступа order и offset ... 226
5.3 Добавление легенды в проект ... 230
Итоги ... 237

6 ***Визуализация распределений*** ... 239
6.1 Группировка данных ... 242
6.2 Рисование гистограммы ... 245
6.3 Создание пирамидальной диаграммы ... 250
6.4 Создание ящичных диаграмм ... 253
6.4.1 Расчет квартилей с помощью квантильной шкалы ... 254
6.4.2 Позиционирование нескольких ящичных диаграмм на графике ... 256
6.4.3 Точечная шкала ... 257
6.4.4 Рисование ящичной диаграммы ... 260
6.5 Сравнение распределений с помощью скрипичных диаграмм ... 265
Итоги ... 278

Часть II НОВЫЕ СТАНДАРТЫ ... 279

7 ***Интерактивные визуализации*** ... 280
7.1 Зачем нужна интерактивность? ... 281
7.1.1 Некоторые лучшие практики интерактивности ... 281
7.2 Фильтрация визуализации ... 282
7.2.1 Обработка пользовательских событий ... 284
7.2.2 Метод classed ... 286
7.2.3 Обновление данных в визуализации ... 287
7.2.4 Создание плавных переходов ... 289
7.3 Отображение дополнительной информации с помощью всплывающих подсказок ... 293
7.3.1 Создание простой подсказки ... 294
7.3.2 Разработка составной подсказки ... 300
7.4 Анимация появления, обновления и исчезновения элементов визуализации ... 307
7.4.1 Построение диаграммы рассеяния ... 309
7.4.2 Фильтрация диаграммы рассеяния ... 313
7.4.3 Создание переходов для повторного использования ... 319
Итоги ... 327

8 ***Интеграция D3 с фреймворками пользовательских интерфейсов*** ... 329
8.1 Подходы к использованию D3 во фреймворках пользовательских интерфейсов ... 331
8.2 Установка библиотеки D3 в проект React ... 333

8.3 Загрузка данных в проект React 335
8.4 Многоразовый подход к контейнерам SVG 338
8.5 Передача части DOM под управление D3 342
8.5.1 React 342
8.5.2 Angular 347
8.5.3 Svelte 349
8.6 Использование D3 как вспомогательной библиотеки 350
8.6.1 React 351
8.6.2 Angular и Svelte 357
8.6.3 Создание кривых 360
8.7 Гибридный подход 366
Итоги 373

9 ***Адаптивные визуализации*** 375
9.1 Что такое адаптивный дизайн? 376
9.1.1 Подход, ориентированный на мобильные устройства 377
9.1.2 Подход, ориентированный на настольные компьютеры 377
9.2 Адаптивный линейный график 378
9.2.1 Адаптация размера текстовых надписей 379
9.2.2 Настройка меток осей 381
9.2.3 Использование минималистского подхода 382
9.3 Адаптивная информационная панель 384
9.3.1 Использование адаптивной сетки 385
9.3.2 Адаптация плотности информации 393
9.3.3 Изменение ориентации диаграммы 398
9.4 Дополнительные советы 399
Итоги 404

10 ***Доступность визуализаций для людей с ограниченными возможностями*** 405
10.1 Как люди с ограниченными возможностями потребляют веб-контент 406
10.2 Соответствие стандартам доступности 407
10.2.1 Текстовая информация 409
10.2.2 Визуальная информация 415
10.2.3 Доступ к экранному диктору 426
10.2.4 Взаимодействия 441
10.2.5 Прочее 445
10.2.6 Дополнительные ресурсы 446
Итоги 451

Часть III СЛОЖНЫЕ ВИЗУАЛИЗАЦИИ ДАННЫХ 452

11 ***Иерархические визуализации*** 453
11.1 Форматирование иерархических данных 457
11.1.1 Файл CSV 457
11.1.2 Файл JSON 461

11.2 Построение упакованной пузырьковой диаграммы....464
11.2.1 Создание макета упакованной пузырьковой диаграммы....465
11.2.2 Рисование упакованной пузырьковой диаграммы....468
11.2.3 Добавление меток....473
11.3 Построение древовидной диаграммы....478
11.3.1 Создание древовидного макета....479
11.3.2 Рисование древовидной диаграммы....481
11.4 Создание других иерархических визуализаций....487
Итоги....492

12 *Сетевые визуализации*....493
12.1 Подготовка сетевых данных....495
12.2 Создание матрицы смежности....497
12.3 Создание дуговой диаграммы....503
12.4 Эксперименты с силами....509
12.4.1 Запуск симуляции действующих сил....509
Итоги....528

13 *Визуализации геопространственной информации*....530
13.1 Географические данные....532
13.1.1 GeoJSON....532
13.1.2 TopoJSON....533
13.2 Рисование карты на основе данных GeoJSON....534
13.2.1 Выбор проекции....534
13.2.2 Улучшение читаемости с помощью картографической сетки....539
13.2.3 Создание картограммы....542
13.2.4 Размещение городов на карте....547
13.3 Масштабирование и панорамирование....554
13.4 Добавление функции выделения мышью....558
13.5 Рисование карты на основе данных TopoJSON....564
13.6 Дополнительные возможности....569
13.6.1 Мозаичные карты....569
13.6.2 Рисование на холсте....569
13.6.3 Растровое проецирование....569
13.6.4 Шестиугольники....569
13.6.5 Диаграммы Вороного....570
13.6.6 Карты-анаморфозы....570
Итоги....575

Часть IV ПРОДВИНУТЫЕ МЕТОДЫ ВИЗУАЛИЗАЦИИ....577

14 *Создание нестандартных визуализаций*....578
14.1 Сбор данных....579
14.2 Изучение данных....583
14.3 Создание эскиза макета....587

14.4 Создание основы проекта 591
14.4.1 Другой подход к созданию адаптивных контейнеров SVG 592
14.4.2 Создание адаптивной сетки SVG 596
14.5 Создание радиальных визуализаций 599
14.5.1 Добавление радиальных осей 599
14.5.2 Применение силовой компоновки к элементам на окружности 605
14.5.3 Рисование радиальной диаграммы с областями 612
14.5.4 Рисование радиальной столбчатой диаграммы 616
14.6 Выбор значимых взаимодействий 619
Итоги 627

15 ***Отображение визуализаций с помощью Canvas*** 628
15.1 Что такое Canvas и когда его использовать 629
15.2 Отображение простых фигур с помощью Canvas 630
15.2.1 Элемент <canvas> 631
15.2.2 Линия 634
15.2.3 Прямоугольник 635
15.2.4 Круг 637
15.2.5 Кривая 638
15.2.6 Текст 639
15.3 Смешанный режим отображения 641
15.4 Стратегия взаимодействия с Canvas 649
Итоги 660

Приложение A. Настройка локальной среды разработки 661
Приложение B. Выбор шкалы 665
Приложение C. Обзор модулей D3 681
Приложение D. Решения упражнений 682
Приложение E. Очень краткое введение в Svelte 705
Предметный указатель 710

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Положительные отзывы ко второму изданию

Эта книга знакомит с инструментами для создания красивых визуализаций данных, двигаясь от простого к сложному.

– Клаудио Родригес (Claudio Rodriguez), Cox Media Group

Лучший справочник по одному из самых мощных инструментов визуализации данных.

– Джонатан Риу (Jonathan Rioux), TD Insurance

Продвигаясь от простеньких примеров до сложных приемов, используемых в реальных проектах, эта книга показывает, как разные компоненты взаимодействуют друг с другом.

– Скотт Мак Киссок (Scott McKissock), USAID

Интереснейшее погружение в мир D3.js.

– Фелипе Вильдосо Касти (Felipe Vildoso Casti), Университет Чили

Предисловие

Как внештатный преподаватель, консультант и дизайнер я хорошо знаком с миром визуализации данных, и мне повезло воочию наблюдать огромное количество изменений в нем. Технологический ландшафт постоянно меняется. Развитие наших инструментов и платформ, с помощью которых плоды нашего труда передаются в руки пользователей, постоянно подталкивает нас вперед.

Когда-то визуализацией данных занималось небольшое сообщество специалистов, но возросшее внимание со стороны ведущих игроков привело к существенному росту как числа энтузиастов, так и их разнообразия. Область визуализации чрезвычайно широка, имеет богатую и разнообразную культуру, и это дает уверенность в дальнейшем ее развитии.

Все расширяющееся сообщество талантливых разработчиков продолжает вносить свежие мысли. Традиционный дискурс и так называемые устоявшиеся убеждения подвергаются сомнению. Повышенная тяга к экспериментам приводит к созданию инновационных методов, влияющих на аудиторию. А границы творческих возможностей простираются за пределы *простых* диаграмм.

Неизменным остается только желание дизайнеров и разработчиков достичь максимальной *выразительности* и *гибкости* при создании визуализаций данных. *Выразительность* открывает доступ к широчайшему набору вариантов представления информации, намного более широкому, чем вы можете себе представить. В течение многих лет D3.js – библиотека для JavaScript – предлагала такую выразительность.

Гибкость помогает решать задачи, слишком сложные или трудоемкие, чтобы решать их вручную. Гибкость сводит к минимуму задержки из-за незнания подходов к решению тех или иных задач и возможно ли их решить в принципе. Гибкость подразумевает знание, когда и почему следует или не следует делать определенный выбор.

Третье издание «D3.js в действии» поможет вам получить и то, и другое. Предыдущие издания давали читателям четкое понимание, *что* может делать библиотека D3.js и *как* она это делает. Но в третьем издании авторы расширили контекст современных практик визуализации данных и добавили ответы на вопросы *когда*, *почему* и для *кого*.

Самые ценные книги в любой дисциплине затрагивают темы, которые могут иметь крутую, а для кого-то и непреодолимую кривую

обучения. Они делают эту кривую более плавной и служат одновременно введением для новичков и хорошим справочником для опытных практиков. Они сплетают воедино кажущуюся объективность технологий с изначально субъективным мастерством визуальной коммуникации. Все это в полной мере относится и к «D3.js в действии».

Эту книгу нужно *использовать*. Слова «в действии» в названии книги отражают ее прикладной и обучающий характер и наличие в ней вдохновляющих примеров и упражнений, помогающих учащимся извлекать новые знания из страниц и применять их на практике.

Эта книга поощряет читателей стремиться к тому, чтобы быть *полезными*. Мысль о том, чтобы что-то делать, потому что вы *хотите*, а не потому что *могли бы*, проходит красной нитью через всю книгу. Вы хотите создавать функционально богатые визуализации? Конечно. Вы хотите создавать их эстетически красивыми? А кто не хочет? Эта книга поможет вам удовлетворить эти желания, но покажет, как делать правильно. Плоды вашего труда будут полезны людям, которым они предназначены.

Книга также делает упор на визуализациях, которые *удобно использовать*. Качественно спроектированная визуализация способна работать со множеством платформ, элегантно адаптируясь к различным формам, размерам и функциям. Чтобы визуализация была удобной, она также должна быть одинаково доступна людям с разными возможностями, и этой, часто игнорируемой теме в книге уделяется должное внимание.

Эта книга, начиная с ее ранних изданий, основана на трудах Элайджи Микса, выдающегося разработчика, видного члена сообщества и вдумчивого критика практик визуализации данных, используемых в отрасли.

Энн-Мари Дюфур оказалась идеальной кандидатурой на роль того, кто возьмется за работу над этим новым изданием. Она обладает редким сочетанием качеств, будучи высококлассным разработчиком систем визуализации данных с тонким чувством эстетики и недюжинными творческими способностями. Ее технические и коммуникативные таланты прекрасно дополняются врожденной способностью разбивать сложные темы на легко усваиваемые и понятные части. Пройдя свой собственный, непростой и долгий путь обучения, Энн-Мари хорошо понимает, как помочь другим, и нет лучшего человека, который мог бы провести вас по этому захватывающему предмету.

Энди Кирк (*Andy Kirk*),
независимый эксперт по визуализации данных

Вступление

В 2017 году я работала разработчиком пользовательских интерфейсов и в какой-то момент поняла, что должна сделать следующий шаг в профессиональном развитии. Мне нравилось разрабатывать веб-сайты, но постоянно чего-то не хватало. Я хотела связать свой опыт разработчика с любовью к преподаванию и моими новыми навыками в программировании. Именно тогда мой партнер предложил мне заняться визуализацией данных. Я не знаю, почему ему показалось, что мне понравится изучать эту бурно развивающуюся область. Когда я загуглила по фразе «визуализация данных», то наткнулась на проект Data Sketches (www.datasketch.es) Надии Бремер (Nadieh Bremer) и Ширли Ву (Shirley Wu). Этот проект произвел на меня сильное впечатление, и я решила, что должна научиться создавать подобные проекты. Именно тогда я впервые познакомилась с библиотекой D3.js, которую они использовали в своей работе.

Я начала понемногу изучать D3, подписываясь на курсы и читая статьи в блогах. Но постепенно стала испытывать все большее разочарование, обнаруживая, что изученные мною фрагменты кода устарели. Мне не нравилось отсутствие простого пошагового подхода к изучению и созданию проектов на D3. Когда началась первая волна пандемии COVID-19 и мир закрылся, у меня наконец появилось время сесть и начать создавать свои первые проекты. Постепенно философия D3 стала обретать смысл и становилась все более и более понятной.

Какое-то время спустя, благодаря дружескому толчку Энди Кирка, со мной связалась Николь Баттерфилд (Nicole Butterfield) из издательства Manning. Она сказала, что они хотят создать обучающий курс по D3, и спросила, интересно ли мне это. Испытывая энтузиазм от своих первых проектов по визуализации данных, я с головой окунулась в эту задачу и начала разрабатывать стратегию изучения D3. В результате через несколько месяцев в свет вышла книга «Interactive Visualization with D3.js» (http://mng.bz/jXrz).

В этом новом издании я хочу предоставить вам, дорогой читатель, свою стратегию создания уникальных и наглядных визуализаций данных. Эта книга и проекты, представленные в ней, помогут вам изучить не только основы, но и продвинутые концепции, и я надеюсь, что она послужит вам справочником на долгие годы.

Благодарности

До работы над этой рукописью я понимала, что написание книги потребует огромного труда, но я не подозревала, сколько людей играют решающую роль в воплощении такого проекта в жизнь. В первую очередь я хочу поблагодарить Николь Баттерфилд (Nicole Butterfield) и Брайана Сойера (Brian Sawyer) из издательства Manning, которые поверили в мою способность создать это новое издание и дали мне зеленый свет. Я также благодарна Элайджу Миксу за то, что он позволил мне продолжить его работу. Я с трудом могу представить себе то странное чувство, когда кто-то, кого ты едва знаешь, делает твое детище своим. Спасибо тебе огромное за доверие!

Огромное спасибо Элеше Хайд (Elesha Hyde), моему редактору-корректору в Manning, и Джону Боргману (Jon Borgman), техническому редактору и старшему инженеру-программисту с более чем 20-летним опытом комплексной разработки. Я тесно сотрудничала с ними обоими в течение последних полутора лет. Они были моими вдохновителями на этом пути, и книга и вполовину не была бы так хороша без их вдумчивых рецензий.

За кулисами над книгой трудилась огромная команда сотрудников издательства Manning. С кем-то мы общались недолго, с кем-то совсем незнакомы. Тем не менее большое спасибо всем вам за вашу самоотверженную работу! Спасибо всем рецензентам: Алену Куньо (Alain Couniot), Алену Ломпо (Alain Lompo), Амогу Рагхунатху (Amogh Raghunath), Арту Бергквисту (Art Bergquist), Эшли Итли (Ashley Eatly), Крису Томасу (Chris Thomas), Деборе Меските (Deborah Mesquita), Елене Гисальберти (Elena Ghisalberti), Эли Рабинович (Eli Rabinovitz), Эсрефу Дурне (Esref Durna), Грегорио Пикколи (Gregorio Piccoli), Хансу Доннеру (Hans Donner), Джеймсу Дж. Байлекки (James J Byleckie), Джереми Аллену (Jereme Allen), Джонатану Бойсеру (Jonathan Boiser), Хуану М. Каррильо де Хеа (Juan M. Carrillo de Gea), Леонарду Грею (Leonard Grey), Марио Руизу (Mario Ruiz), Майклу Брайту (Michael Bright), Патрису Мальдагу (Patrice Maldague), Пьерфранческо д'Орсонье (Pierfrancesco D'Orsogna), Прасанту Расаму (Prasanth Rasam), Родни Вайсу (Rodney Weis), Саймону Верховену (Simon Verhoeven), Шрираму Мачарле (Sriram Macharla), Тамижу Арасу С. (Thamizh Arasu S.) и Ивесу Дорфсману (Yves Dorfsman) – ваши предложения помогли сделать эту книгу лучше.

Также я хочу поблагодарить моих друзей и коллег по визуализации данных: Инбала Рифа (Inbal Rief), Седрика Шерера (Cédric Scherer)

и Георгиоса Караманиса (Georgios Karamanis) – вы были свидетелями всех самых важных этапов в моей карьере, от «я получила работу!» до многочисленных «я почти дошла до конца… еще чуть-чуть». Ваши отзывы о книге, о моей работе и ваш постоянный энтузиазм вдохновляли меня на протяжении всего пути. Кроме того, спасибо Энди Кирку (Andy Kirk), который, по своей неизменной доброте, согласился написать предисловие к этой статье. Мы все считаем вас одним из ведущих наставников в сфере визуализации данных. Спасибо за все, что вы делаете для нашего ремесла!

Наконец, как оказалось, мне не удалось найти здоровый баланс между работой и личной жизнью во время работы над этим проектом, и я не смогла закончить его без своей семьи. Адам, ты моя опора. Спасибо за поддержку во всех моих безумных начинаниях! Альберт и Арно, спасибо за ваши объятия и поцелуи в течение тех долгих часов, которые я проводила за своим столом. Надеюсь, вам понравится читать эту книгу, зная, что были рядом со мной, когда я ее писала.

О книге

Кому адресована эта книга

Эта книга адресована всем, кто хочет полной творческой свободы, занимаясь визуализацией данных, от настраиваемых классических диаграмм до уникальных макетов. Вы можете быть опытным специалистом по анализу данных, журналистом, дизайнером или просто энтузиастом. Вы сделали правильный шаг, решив изучить D3.js! Вы быстро поймете, что это мудрое вложение вашего времени. Освоив D3, вы откроете для себя уровень свободы и потенциал для творчества, не имеющий равных среди бесчисленного множества инструментов визуализации данных, доступных сегодня.

Как организована эта книга

Возможно, вы знаете, что D3 принадлежит экосистеме инструментов разработки пользовательского интерфейса: HTML, CSS и JavaScript. Прежде чем погрузиться в главу 2, я раскрою основы этих инструментов и как все вместе они применяются для создания красивых и интерактивных веб-страниц. Вам не нужно быть экспертом, но немного предварительных знаний сделают изучение D3 намного более простым. Если вы хотите освежить свои навыки разработки пользовательского интерфейса, мы рекомендуем обратиться к следующим ресурсам:

- *2023 Web Development Bootcamp* Максимилиана Шварцмюллера (Maximilian Schwarzmüller) (http://mng.bz/WEe4);
- *Complete Intro to Web Development* на сайте Frontend Masters (http://mng.bz/8wyZ).

Объяснение основной темы в этой книге ведется последовательно, от простого к сложному. Часть I охватывает основы, такие как работа с данными и создание простых диаграмм, а часть II фокусируется на удовлетворении новых ожиданий и создании интерактивных и адаптивных проектов с применением D3 и фреймворков JavaScript, таких как React и Svelte. Каждая глава в части III охватывает более продвинутые приемы визуализации данных: иерархии, сети и карты. Наконец, в части IV мы пригласим вас за кулисы создания уникального макета визуализации, а также обсудим производительность и проблемы объединения D3 с Canvas.

Если вы только приступаете к изучению D3, то мы рекомендуем прочитать главы с 1 по 7 по порядку. Они помогут вам построить ментальную модель D3 и освоить все строительные блоки, прежде чем приступить к более сложным темам. D3 имеет плохую репутацию из-за крутой кривой обучения, но эти первые главы помогут вам лучше понять эту библиотеку.

Если вы уже знакомы с D3 и хорошо разбираетесь в основах, то можете выбирать главы по своим интересам. Главы с 8 по 15 посвящены конкретным идеям и типам диаграмм, и их стоит прочитать. Сведения из этих глав обязательно пригодятся вам в ваших проектах.

Кроме того, всем читателям мы настоятельно рекомендуем не только ПРОЧИТАТЬ книгу, но и ВЫПОЛНИТЬ все примеры и упражнения. Каждая глава описывает определенный проект(ы) визуализации данных, и, внимательно исследуя их, вы научитесь интегрировать концепции, объясняемые в тексте. Применение этих концепций на практике будет иметь решающее значение и сделает ваш путь обучения более плавным и быстрым.

Исходный код каждого проекта и упражнения, а также решений доступен в репозитории книги на GitHub (http://mng.bz/Xqjv). Решения упражнений приводятся и в приложении D. Все файлы и фрагменты кода в этой книге используют D3 версии 7 – последней на момент выхода книги. Для редактирования и опробования кода вам понадобятся только текстовый редактор и браузер. Мы рекомендуем VS Code и Chrome или Firefox.

О примерах кода

Книга содержит много примеров исходного кода как в пронумерованных листингах, так и в тексте. В обоих случаях исходный код оформлен моноширинным шрифтом, чтобы его легко было отличить от обычного текста. Кое-где код также выделяется жирным шрифтом, чтобы подчеркнуть изменения по сравнению с предыдущими шагами, например когда в уже описанный код добавляется что-то новое.

Во многих случаях исходный код был переформатирован; мы добавили переносы строк и отступы, чтобы уместить код по ширине книжной страницы. В редких случаях даже этого было недостаточно, и кое-где мы включили в листинги маркеры продолжения строки (➥). Кроме того, некоторые комментарии были удалены из листингов, если код описывается в тексте. Многие листинги сопровождаются дополнительными аннотациями, выделяющими важные концепции.

Выполняемые фрагменты кода можно получить из liveBook (онлайн) версии этой книги, доступной по адресу https://livebook.manning.com/book/d3js-in-action-third-edition. Код примеров из книги доступен также для загрузки на веб-сайте Manning по адресу www.manning.com/books/d3js-in-action-third-edition и на GitHub по адресу http://mng.bz/Xqjv.

Об авторах

Элайджи Микс (**Elijah Meeks**) занимается визуализацией данных вот уже более 20 лет и работал в некоторых из самых динамичных организаций в мире, включая Stanford, Netflix и Apple. Он стал соучредителем Notable – стартапа, ориентированного на разработку вычислительных блокнотов, инструментов бизнес-аналитики и ИИ. Основал и в настоящее время является директором по публикациям Data Visualization Society – профессионального общества, занимающегося развитием средств визуализации данных. Помимо этого, в настоящее время он занимает пост главного инженера в Confluent.

Элайджи – автор многих книг и статей по теме визуализации данных, в том числе и этой книги «D3.js в действии», и создатель серии видеороликов *Pearson Designing for the Data Visualization Lifecycle*. Он является создателем таких библиотек, как Semiotic, а также интерактивных решений, таких как *ORBIS: The Transportation Network Model of the Roman Empire*.

Энн-Мари Дюфур (**Anne-Marie Dufour**) имеет большой опыт в области машиностроения, вычислительной гидродинамики и разработки пользовательских интерфейсов. Специализировалась на визуализации данных, стремясь объединить свою любовь к науке, программированию, дизайну и преподаванию. Этот уникальный набор навыков помог ей развить четкое понимание, как визуализация данных может помочь понять сложные явления и как воплотить их в современные веб-приложения. В настоящее время работает инженером по визуализации данных в PingThings.

Преподавая различные инженерные предметы, такие как гидродинамика, теплотехника и экологические и социальные последствия инженерных проектов, она достигла глубокого понимания, как структурировать информацию и представлять ее в легко усвояемой форме, чтобы поддерживать у студентов желание учиться. Недавно она создала серию проектов *Interactive Visualization with D3.js* для Manning Publications.

Об иллюстрации на обложке

Иллюстрация на обложке книги называется «Типичный мавританский паломник, возвращающийся из Мекки в 1586 году». Она взята из книги Томаса Джеффериса, опубликованной между 1757 и 1772 годами.

В те времена по одежде легко было определить место жительства, профессию и социальное положение. Издательство Manning прославляет изобретательность и инициативность компьютерного бизнеса с помощью подобных иллюстраций на обложках своих книг, демонстрирующих богатое разнообразие региональной культуры много веков назад.

Часть I

Основы D3.js

Добро пожаловать в мир D3.js! Мы знаем, что вам не терпится начать создавать умопомрачительные визуализации данных. Очень скоро вы займетесь этим, обещаем! Но сначала рассмотрим некоторые основы, без знания которых не обойтись. Понятия, с которыми вы столкнетесь в этой части, неоднократно будут встречаться вам в вашем путешествии по D3, и их понимание даст вам определенное преимущество, когда вы подойдете к созданию сложных визуализаций.

В главе 1 мы поговорим о преимуществах библиотеки D3 и ее экосистемы. Мы также познакомим вас с концепциями, которые помогут вам в обучении, такими как использование масштабируемой векторной графики (Scalable Vector Graphics, SVG), манипулирование данными с помощью JavaScript и объединение вызовов методов в цепочки.

Затем, в главах 2 и 3, мы с помощью D3 создадим нашу первую визуализацию данных: столбиковую диаграмму. Но перед этим мы расскажем, как манипулировать объектной моделью документа (Document Object Model, DOM) и работать с данными. В главах 4 и 5 мы построим более сложные визуализации, используя генераторы форм и макетов из D3. В главе 6, завершающей эту часть, мы обсудим тему распределений, с которой рано или поздно сталкиваются все специалисты по визуализации данных.

1 Введение в D3.js

https://t.me/it_boooks/2

В этой главе рассматриваются:

- роль D3.js и философия, лежащая в ее основе;
- инструменты, используемые вместе с D3.js для создания визуализаций данных;
- создание и стилизация масштабируемой векторной графики (SVG) программным способом;
- лучшие практики применения D3.js для визуализации данных.

Учитывая изобилие инструментов визуализации данных, появившихся за последнее десятилетие, возникает вопрос: стоит ли изучение D3 таких хлопот. Ответ прост: изучение D3 – это надежное вложение сил и времени. Да, кривая обучения может быть крутой и требовать упорства и самоотверженности, но вы не только обретете возможность создавать традиционные диаграммы, которые можно создавать с помощью других библиотек и настраивать их по своему желанию, но и получите свободу пойти своим путем и создавать визуализации, которые действительно соответствуют вашим данным и аудитории.

D3.js – это библиотека, лежащая в основе многих захватывающих визуализаций данных в интернете. Это инструмент, дающий полную

творческую и техническую свободу в визуализации данных, независимо от того, создаете ли вы интерактивные прототипы для исследований, адаптивные панели управления данными или прокручиваемые истории с данными, такие как «Почему Будапешт, Варшава и Литва разделились на две части» Мартена Ламбрехтса (Maarten Lambrechts), изображенная на рис. 1.1.

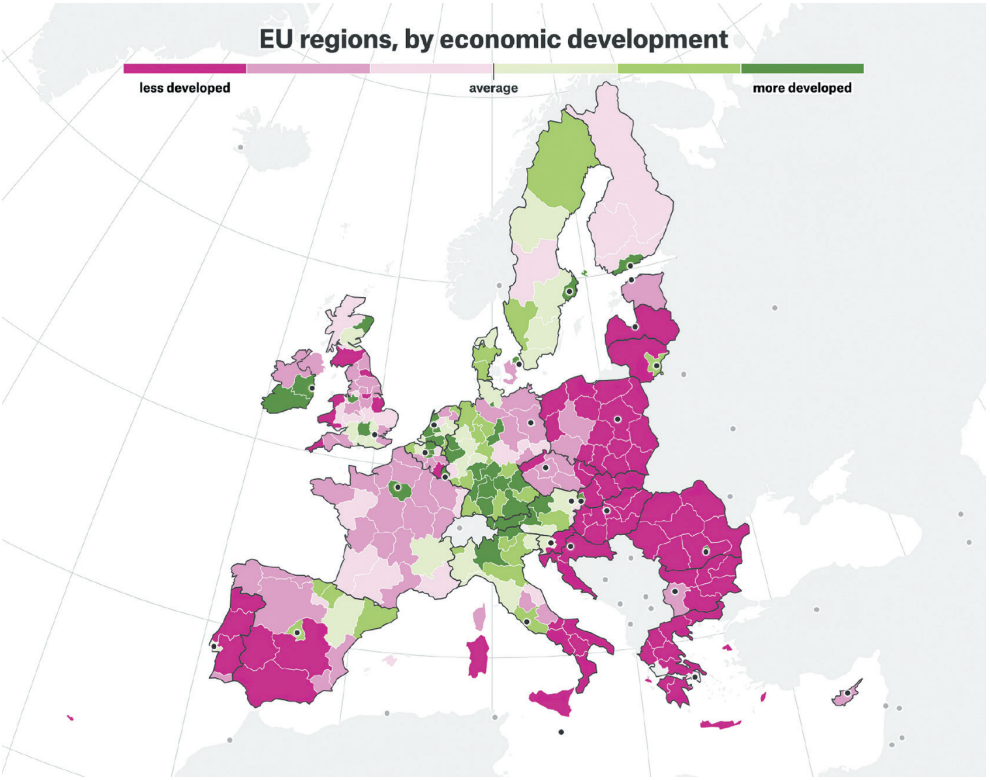


Рис. 1.1 Разработчики D3 имеют доступ к широкому спектру представлений данных, таких как карты. Вот пример визуализации, созданной Мартеном Ламбрехтсом (https://pudding.cool/2019/04/eu-regions/**)**

В этой первой главе мы познакомимся с экосистемой D3 и несколькими концепциями, такими как графика SVG и методы манипулирования объектами JavaScript, крайне важными для цифровой визуализации данных.

ПРИМЕЧАНИЕ На протяжении всей книги мы будем использовать названия D3.js и D3 как взаимозаменяемые.

1.1 Что такое D3.js?

Название D3.js расшифровывается как *Data-Driven Documents* (документы, управляемые данными). Это библиотека JavaScript с открытым исходным кодом, написанная в 2011 году Майком Бостоком (Mike Bostock) для создания динамических и интерактивных визуализаций данных. С тех пор появилось много других библиотек визуализации данных, но многие из них основаны на D3 благодаря ее мощности и гибкости.

1.1.1 Необходимость в визуализации данных для веба

D3 разрабатывалась для удовлетворения острой потребности в сложных визуализациях данных для веба. Представьте, что ваша компания использует инструмент бизнес-аналитики, но он не показывает закономерности в данных, которые нужны вашей команде аналитиков. Они просят вас создать визуализацию, которая точно отражает поведение ваших клиентов и адаптирована для вашей конкретной области. Эта визуализация должна быть быстрой, интерактивной и доступной для совместного использования несколькими специалистами организации. D3 – естественный выбор для такой задачи.

Или представьте, что вас наняли для создания развертываемой диаграммы, визуализирующей тенденции соблюдения прав национальных меньшинств в последние десятилетия по всему миру. Такая диаграмма должна содержать множество визуализаций, трансформирующихся по мере ее развертывания пользователем, раскрывающих больше информации по событиям от мыши и адаптирующихся к размеру экрана. D3 – это правильный инструмент для создания такого проекта.

Майк Бостоk (Mike Bostock) первоначально создавал D3, чтобы получить возможность использовать преимущества новых веб-стандартов, которые, как он выразился, «помогают избежать использования проприетарных представлений и обеспечивают исключительную гибкость, раскрывая все возможности таких стандартов, как CSS3, HTML5 и SVG» (http://d3js.org). D3.js версии 7, которая была самой свежей на момент написания книги, продолжает эту тенденцию, стремясь обеспечить совместимость с модулями ECMAScript для упаковки кода JavaScript и возможность повторного использования.

D3.js предоставляет разработчикам возможность создавать насыщенные интерактивные проекты в форме традиционного веб-контента, как, например, «The Inside Scoop of Ben & Jerry's» на рис. 1.2. Это делает их более переносимыми, масштабируемыми и простыми в поддержке большими группами, где другие члены команды могут не знать конкретный синтаксис D3.

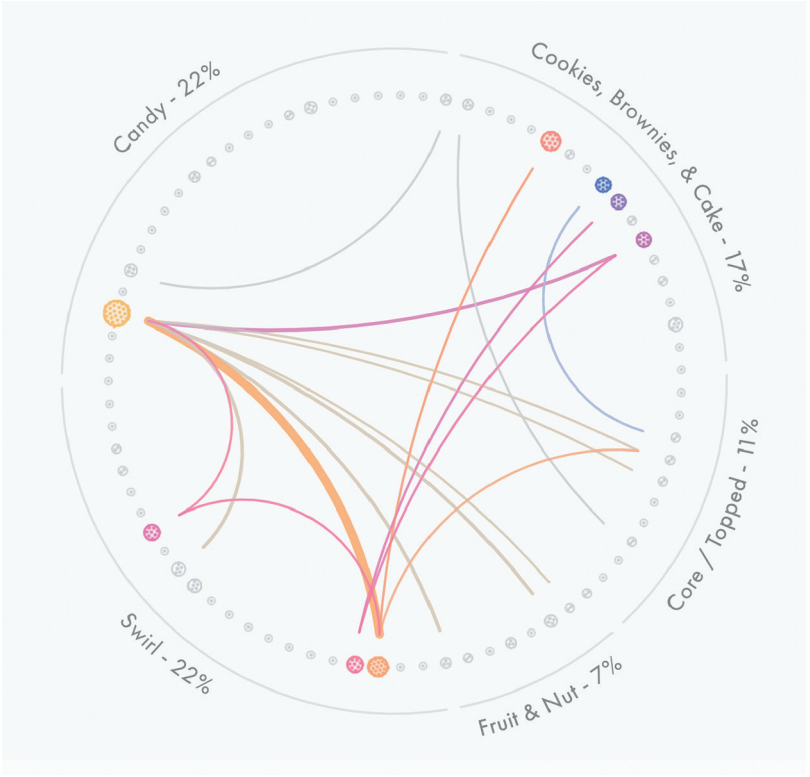


Рис. 1.2 D3 – это низкоуровневая библиотека, дающая полную техническую и творческую свободу. Эта хордовая диаграмма является частью проекта «The Inside Scoop of Ben & Jerry's» Хешама Эйссы (Hesham Eissa) и Линдси Поултер (Lindsey Poulter) (https://benjerry .heshlindsdataviz.com/**)**

Решение Bostock сосредоточиться на данных и создать библиотеку, способную представлять карты так же легко, как линейные и сетевые диаграммы, также означает, что разработчику не придется изучать абстракции и синтаксис множества библиотек – одной для карт, другой для рисования фигур SVG и еще одной для отображения сетевых диаграмм. Фактически программный код, запускающий интерактивную визуализацию сетей с использованием D3, близок к чистому JavaScript и похож на код, отображающий динамические точки на карте D3. Во всех случаях используются одни и те же методы, но данные могут быть сформированы одним способом для представления в форме сети и другим – для геопространственного представления на карте. Кривая изучения этой библиотеки намного круче, чем в случае с другими инструментами, однако силы и время, потраченные на изучение D3, окупятся сторицей.

D3 не только способна создавать сложную и разнообразную графику, но и поддерживает высокий уровень интерактивности, благосклонно воспринимаемый пользователями, что имеет решающее

значение для современной веб-разработки. В D3 каждый элемент каждой диаграммы, от вращающегося глобуса до части круговой диаграммы, можно сделать интерактивным. И поскольку D3 написана человеком, хорошо разбирающимся в практике визуализации данных, она включает многие интерактивные компоненты и поведение, такие как выбор узлов сети (рис. 1.3), которые считаются стандартными как для визуализации данных, так и для веб-разработки.

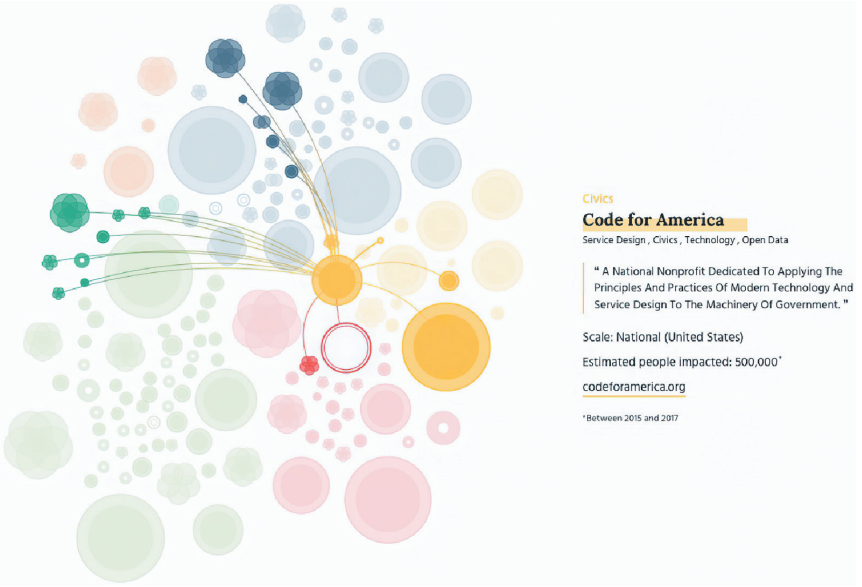


Рис. 1.3 Интерактивность – это главная особенность D3. В этой визуализации сети манипуляции мышью выявляют связи между различными организациями и информацию, относящуюся к выбранному узлу (http://mng.bz/QROG)

1.1.2 Когда следует использовать D3.js?

Сфера визуализации данных переживает бум популярности, и количество инструментов для создания графики, связанной с данными, за последнее десятилетие резко возросло. У нас есть инструменты бизнес-аналитики, такие как Microsoft Excel, широко используемый для визуализации данных, и Power BI, решение от Microsoft для создания дашбордов – панелей мониторинга. С другой стороны, специалисты по данным все чаще обращаются к таким средствам визуализации, как ggplot2 для R и Matplotlib для Python.

Также на передний план вышли инструменты «укажи и щелкни» на основе браузера, такие как Tableau, Flourish, Datawrapper и RAWGraphs, позволяющие создавать потрясающие визуализации с минимальными техническими знаниями.

Наконец, появились библиотеки для JavaScript, такие как Highcharts, Chart.js и Plotly, специализирующиеся на создании интерактивных веб-визуализаций. И этот список далеко не полный.

Так какое же место в этом океане инструментов визуализации данных занимает D3? Когда и как она используется? Мы можем сказать, что с использованием D3 можно создавать любые диаграммы, которые можно создать с помощью других упомянутых библиотек, но обычно D3 – не лучший выбор для построения простых традиционных диаграмм или для этапа исследований, когда выбирается тип визуализации, лучше всего подходящий для представления имеющихся данных. Для создания проектов D3 требуется время, и D3 действительно блистает в сложных, интерактивных и оригинальных проектах. Визуализация данных не ограничивается линейными диаграммами и диаграммами рассеяния! Только что упомянутые инструменты часто фокусируются на предопределенных диаграммах, тогда как D3 позволяет привязать данные к любому графическому элементу и создать что-то новое, например музыкальное сопровождение (рис. 1.4), комбинируя визуальные элементы уникальными способами. Мы используем D3, потому что хотим мыслить нестандартно и не хотим ограничиваться тем, что предлагает библиотека.

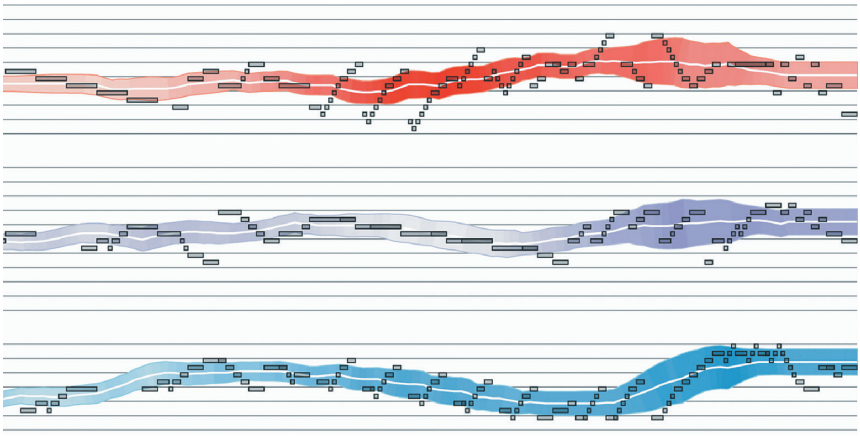


Рис. 1.4 В D3 имеются функции рисования векторной и обычной графики, позволяющие создавать уникальные визуализации, такие как это представление музыкальных партитур, созданное Элайджи Миксом

Вот пример использования D3 в проекте визуализации данных. Сначала мы начинаем с уже существующего набора данных или с данных, собранных вручную. Обычно мы тратим значительное количество времени на предварительную обработку, форматирование и подготовку данных к анализу. Для этой цели прекрасно подходят такие языки программирования, как Python и R. Они могут помочь нам извлечь информацию, скрытую в данных. Для простой обработ-

ки и анализа данных также можно использовать Excel, не требующий обладания техническими знаниями. Для базового исследования данных можно даже использовать JavaScript и D3, поскольку они предлагают поддержку статистических методов, которые мы обсудим далее в этой книге.

После начала процесса анализа данных обычно создается несколько прототипов, помогающих уточнить направление для дальнейшего движения. Такие инструменты, как Tableau и RAWGraphs, позволяют быстро создавать подобные прототипы. Это очень важный шаг, и на этом этапе обычно создаются простые и незатейливые визуализации. Мы стараемся не привязываться к нашим идеям на этапе прототипирования и не тратить на них много времени. Нередко нам приходится отказываться от каких-то наработок и начинать все сначала, пока мы не найдем наиболее подходящее решение. Сетевые диаграммы являются исключением, и если проект предполагает их использование, то тогда лучше сразу взять за основу D3.

Наконец, выбрав желаемый тип визуализации, например диаграмму Сэнки (рис. 1.5), мы приступаем к реализации и программируем ее с помощью D3! В настоящее время этот шаг реализации, включающий программирование, особенно характерен для одностраничных приложений и нередко предполагает использование таких фреймворков, как React или Svelte.

1.1.3 Как работает D3.js

Вы, возможно, уже пробовали использовать D3 и обнаружили, что это очень непросто. Возможно, вы ожидали, что эта библиотека будет работать подобно другим библиотекам диаграмм. Примером может служить создание столбчатой диаграммы, как описывается в главах 2 и 3. В D3 нет какой-то одной функции для создания столбчатой диаграммы. Вместо этого в библиотеке имеется функция, добавляющая контейнер `<svg>` в объектную модель документа (Document Object Model, DOM), и отдельный набор функций, добавляющих элементы `<rect>` для каждой точки данных. Затем для вычисления длин прямоугольников, составляющих гистограмму, мы используем шкалы и задаем их атрибуты `height`. Наконец, мы используем третий набор функций, добавляющий оси `x` и `y`.

Как можно видеть на рис. 1.6, это гораздо более длительный процесс, чем использование специализированной библиотеки диаграмм, такой как Highcharts. Но явный способ работы с данными и графикой в D3 также является его сильной стороной. Другие библиотеки диаграмм позволяют быстро создавать линейные и круговые диаграммы, но они часто оказываются бессильными, когда возникает необходимость создать визуализацию, выходящую за рамки традиционных диаграмм, или когда требуется организовать взаимодействия с поль-

зователем. Это не относится к библиотеке D3, которая позволяет создавать любые диаграммы, управляемые данными, и реализовать любую интерактивность, которую только можно представить.

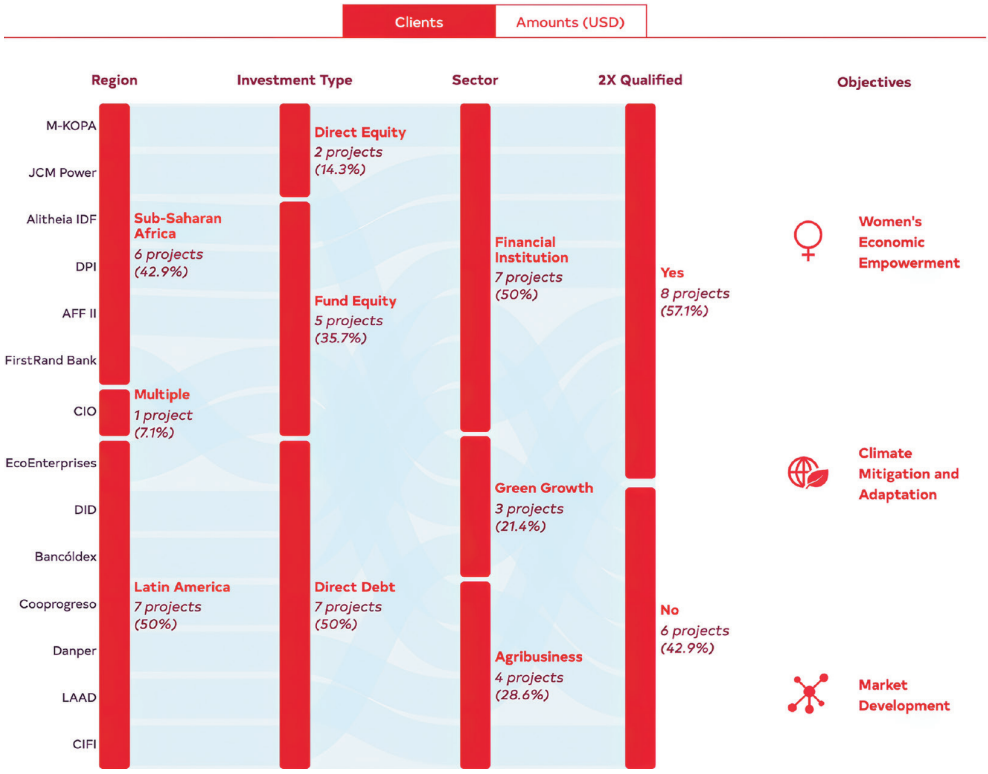
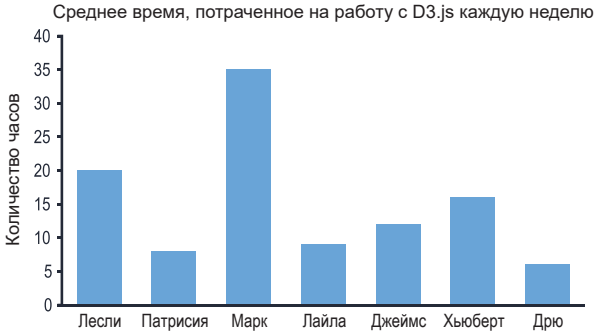


Рис. 1.5 «Собрание инклюзивных предприятий для устойчивого будущего» – отмеченный наградами проект, созданный Voilà (https://chezvoila.com/project/findevportfolio/)

На рис. 1.7 представлена ментальная карта нашего обычного подхода к созданию визуализаций данных с помощью D3. Мы начинаем с организации набора данных, например в виде файла CSV или JSON, и используем модуль d3-fetch для его загрузки в проект. Затем мы обычно выполняем некоторые манипуляции, чтобы отформатировать данные. Например, мы приводим числа и даты к единому формату. После этого исследуем набор данных с целью выявить его основные характеристики. Например, часто полезно знать максимальные и минимальные значения. После этого мы приступаем к созданию визуализации, для чего комбинируем различные функции D3, с которыми вы познакомитесь в этой книге. В заключение мы добавляем интерактивность, обрабатывая события от мыши, и даем пользователям возможность фильтровать данные или увеличивать масштаб визуализации.



Highcharts

```
Highcharts.chart('container',
{
 chart: {
  type: 'column'
 },
 xAxis: {
  categories: names
 },
 yAxis: {
  title: {
   text: 'Number of hours'
  }
 },
 series: [{
  data: [20, 8, 35, 9, 12,
         16, 6]
 }]
}
);
```

D3.js*

```
d3.csv('../data/my_data.csv')
 .then(data => {
  createBarChart(data);
 });
```

Загрузка данных

```
const createBarChart = (data) => {
 const xScale = d3.scaleBand()
  .domain(names)
  .range([0, width]);
 const yScale = d3.scaleLinear()
  .domain([0, 40])
  .range([height, 0]);
```

Создание шкал

```
 const svg = d3.select('#viz')
  .append('svg')
   .attr('width', width)
   .attr('height', height);
```

Добавление родительского элемента SVG в DOM

```
 const bars = svg
  .selectAll('rect')
  .data(data)
  .join('rect')
```

Привязка данных к прямоугольникам

Размещение прямоугольников и установка их размеров

```
   .attr('x', d => xScale(d.name))
   .attr('y', d => yScale(+d.num_hours))
   .attr('width', 60)
   .attr('height', d => 400 -
         yScale(+d.num_hours))
   .attr('fill', '#6ba5d7');
```

```
 const xAxis = svg
  .append('g')
  .call(d3.axisBottom(xScale));
 const yAxis = svg
  .append('g')
  .call(d3.axisLeft(yScale));
};
```

Добавление осей

* Мы упростили код D3.js для краткости

Рис. 1.6 Создание столбчатой диаграммы с помощью Highcharts и D3.js. Код Highcharts проще и короче, но D3.js дает больше возможностей

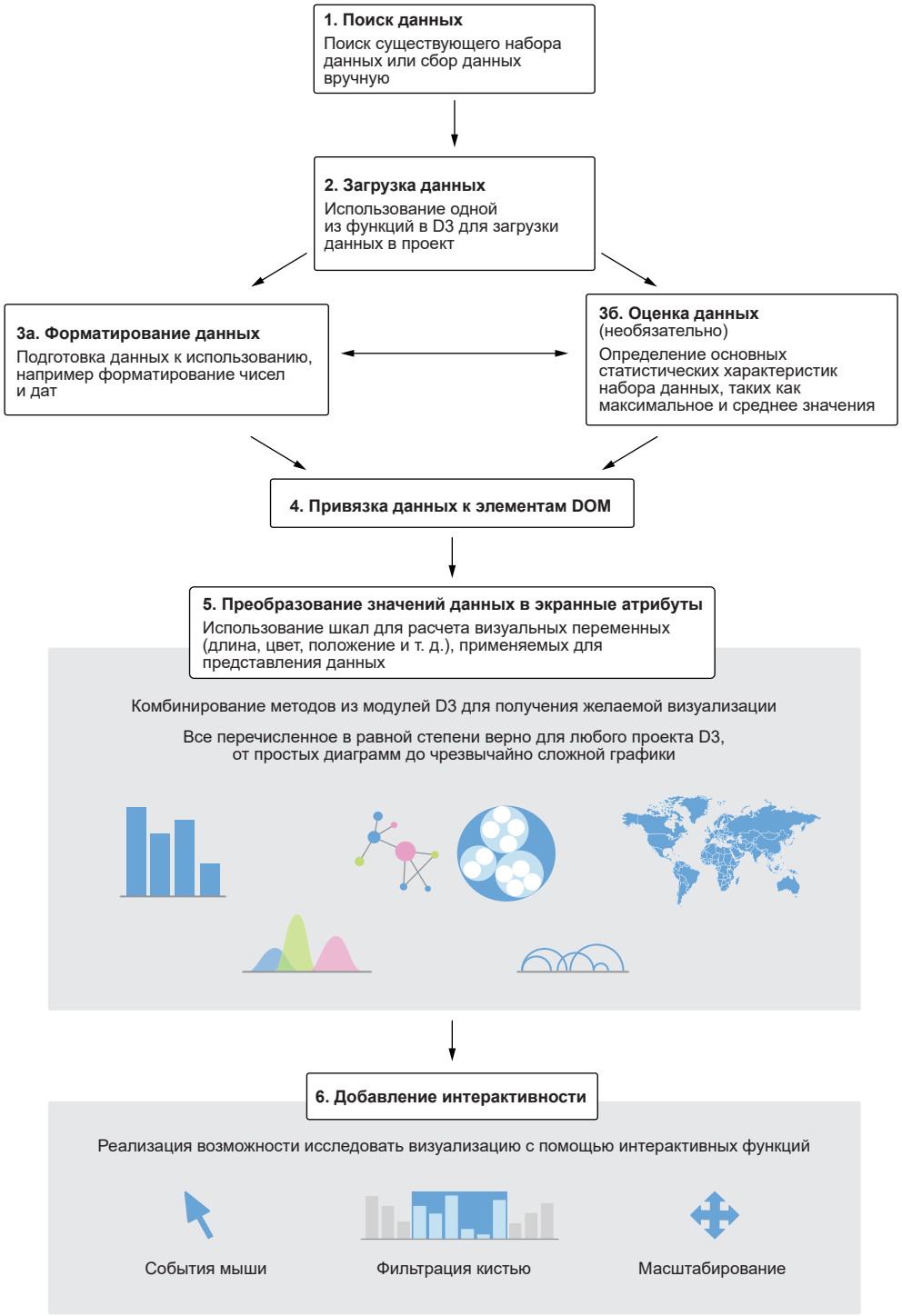


Рис. 1.7 Подход к созданию визуализаций данных с помощью D3.js

Интервью с Элайджи Микс

Микс – инженер по визуализации данных и автор первых двух изданий книги «D3.js в действии».

Расскажите немного о своем опыте и о том, как вы пришли в сферу визуализации данных.

Я занимался визуализацией данных в той или иной форме почти 15 лет в таких организациях, как Apple, Netflix, Data Visualization Society, Stanford и Notable – в стартапе, в основании которого я принимал участие. В визуализацию данных я пришел нетрадиционным путем. Сначала я занялся географической информационной системой (ГИС) для моей докторской диссертации о формировании государства в раннем Китае. Затем я занялся сетевой визуализацией и уже потом начал создавать столбчатые и линейные диаграммы.

Как вы открыли для себя D3 и что подтолкнуло вас заняться освоением этой библиотеки?

В начале карьеры для создания интерактивных приложений визуализации данных я использовал Flash и ActionScript3. Когда стало ясно, что Flash прекратил развитие, я начал изучать библиотеку Protovis, которую быстро заменил на D3. Чем глубже я изучал D3, тем отчетливее понимал суть визуализации данных. Более того, я обнаружил, что D3 помогла мне узнать больше о структурах данных, аналитике и машинном обучении.

Вы написали первые два издания «D3.js в действии» в ту пору, когда было очень мало ресурсов, связанных с D3. Как этот проект воплотился в жизнь?

Я всегда буду благодарен издательству Manning за предоставленную возможность написать эти книги. Работа над ними заставила меня глубоко и всесторонне изучить те возможности D3, которые я просто не использовал в своих проектах. Писательский труд научил меня более тщательно подходить к планированию содержания книги. Первое издание включало примеры, где D3 использовалась почти как MVC для создания HTML-контента, обработки пользовательских событий и разработки макетов, генераторов и компонентов. Во втором издании я отказался от части этого содержания в пользу более практических сведений, таких как интеграция D3 с React.

За последнее десятилетие вы серьезно продвинулись по картерной лестнице, начав с должности первого инженера по визуализации данных в Netflix, а затем став ИТ-директором компании Noteable. За это время вы видели значительные преобразования в отрасли. Что вы думаете о развитии визуализации данных на протяжении многих лет и куда, по вашему мнению, она движется?

Я твердо верю, что любая культура развивается скачкообразно, и практика визуализации данных не исключение. Мы начинали с компьютеров, предоставляющих нам простые возможности построения диаграмм для визуализации табличных данных (например, диаграмм в Excel).

Когда появилась D3, энергия сообщества была направлена на изучение этой библиотеки и приемов ее использования для создания все более сложных и динамичных диаграмм. Теперь мы находимся в точке, когда существует достаточно примеров визуализации данных в разных областях (будь то бизнес-аналитика, журналистика или наука о данных), поэтому ожидания от создаваемых продуктов визуализации данных выходят за рамки того, что в более ранние периоды казалось очень отчетливым. Мы наблюдаем смещение фокуса на создание более интегрированных решений визуализации данных, которые позволяют разным людям с разными навыками и ожиданиями работать над одними и теми же продуктами, управляемыми данными.

1.2 Экосистема D3: что нужно знать для начала работы

D3.js никогда не используется отдельно, она является частью экосистемы технологий и инструментов, которые мы объединяем для создания насыщенных веб-интерфейсов. Подобно любым веб-страницам, проекты D3 создаются в DOM и используют мощь HTML5. Да, D3 может создавать традиционные элементы HTML, такие как разделы (`<div>`) и списки (`<ul>`, `<ol>`), и манипулировать ими, но чаще мы создаем наши визуализации с помощью графики SVG или в Canvas, элементе HTML, который отображает растровые изображения, создаваемые сценариями. Мы также можем использовать старые добрые таблицы стилей CSS, которые способны улучшить проекты D3 и сделать их более простыми в обслуживании, особенно в больших командах.

Учитывая, что D3 – это библиотека JavaScript, мы, естественно, стремимся объединять методы D3 с собственными функциями JavaScript для доступа к данным и управления ими. В настоящее время D3 полностью поддерживает ECMAScript 2015 (ES6-версия JavaScript) и большинство последних усовершенствований. D3 также поставляется в виде модулей, которые можно интегрировать в фреймворки и библиотеки, предназначенные для создания веб-проектов (React, Svelte и т. д.). Использование этих модулей часто оказывается более предпочтительным, потому что при этом не загрязняется глобальная область видимости наших приложений.

В этом разделе мы кратко обсудим перечисленные технологии и их роль в экосистеме D3. Знание SVG является основополагающим для понимания D3, поэтому мы особенно подробно исследуем основы, которые вы должны усвоить, чтобы начать создавать визуализации. Если вы уже знакомы с HTML, SVG, CSS, JavaScript и модулями

JavaScript, то можете просто бегло просмотреть этот раздел или сразу перейти к разделу 1.3.

1.2.1 HTML и DOM

Когда вы открываете веб-страницу, то первым делом браузер загружает файл HTML, такой как в следующем примере. Браузер анализирует HTML-файл и строит дерево DOM – программный интерфейс веб-документов. Мы часто называем его деревом DOM, потому что оно состоит из набора вложенных элементов, также называемых узлами, или тегами. В нашем примере элементы `<head>` и `<body>` являются дочерними элементами родительского элемента `<html>`. Аналогично тег `<body>` является родительским элементом для тегов `<h1>`, `<div>` и `<p>`. Заголовок `<h1>` также является братским элементом для элемента `<div>`. После загрузки веб-страницы вы видите на экране элементы, содержащиеся в теге `<body>`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Простой HTML-файл | D3.js в действии</title>
  </head>
  <body>
    <h1>Это заголовок</h1>
    <div>
      <p>Это абзац.</p>
      <p>Это еще один абзац.</p>
    </div>
  </body>
</html>
```

В DOM о каждом элементе имеется три категории информации, определяющей его поведение и внешний вид: стили, атрибуты и свойства. *Стили* определяют цвет, размер, границы, прозрачность и т. д. *Атрибуты* включают классы и идентификаторы, хотя некоторые атрибуты также могут определять внешний вид, в зависимости от типа элемента. Для элементов SVG атрибуты используются для определения местоположения, размера и пропорций различных фигур. *Свойства* обычно определяют состояние, как, например, свойство «checked» флажка, которое имеет значение true, если флажок установлен, и false, если флажок не установлен. Термины «атрибут» и «свойство» часто используются взаимозаменяемо, но вообще это две разные вещи. Атрибут определяет начальное состояние при отображении DOM, а свойство – текущее состояние элемента, которое может изменяться по мере взаимодействия

пользователя с интерфейсом. В главе 2 мы обсудим методы D3, используемые для генерации или изменения стилей и атрибутов элементов HTML и SVG.

DOM также определяет порядок отрисовки элементов на экране, при этом дочерние элементы отрисовываются после родительских элементов и внутри них. CSS-свойство `z-index` позволяет отчасти управлять порядком отрисовки традиционных элементов HTML, но элементы SVG отрисовываются строго в том порядке, в каком они появляются в DOM, подобно тому, как художник рисует одни элементы поверх других.

1.2.2 Масштабируемая векторная графика

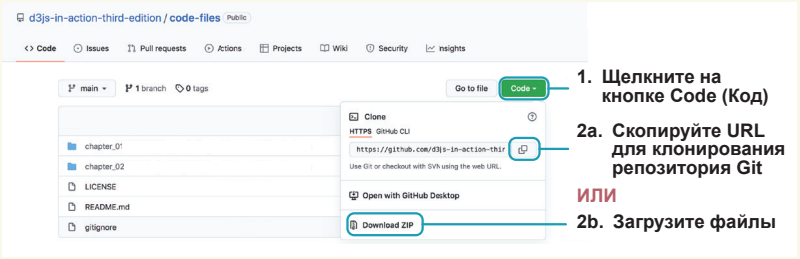
Появление масштабируемой векторной графики (Scalable Vector Graphics, SVG) изменило облик интернета, и это не фигура речи. За несколько лет SVG стал основным инструментом веб-разработки. Если растровая графика (PNG и JPG) состоит из крошечных пикселей, которые становятся видимыми при большом увеличении, векторная графика создается с помощью математики и геометрии. Она сохраняет четкий вид при любом размере и разрешении экрана. Еще одним значительным преимуществом SVG является возможность вводить ее непосредственно в DOM, что позволяет разработчикам манипулировать своими элементами и делать их доступными для устройств чтения с экрана. При правильном построении графика SVG также оказывается более экономичной, так как размер файла с графикой намного меньше размеров эквивалентных растровых изображений.

При создании визуализаций данных с помощью D3 мы обычно внедряем фигуры SVG в DOM и изменяем их атрибуты для получения визуальных элементов. Понимание особенностей работы SVG, основных ее фигур и их атрибутов, определяющих внешний вид, имеет важное значение для большинства проектов D3. В этом разделе мы рассмотрим фигуры SVG, которые вы будете использовать снова и снова в своих проектах D3. Если вы незнакомы с SVG, то обязательно опробуйте все представленные нами примеры и выполните упражнения. Мы обещаем, что это значительно облегчит работу с D3 в будущем.

Как получить файлы с примерами кода

Каждая глава в этой книге включает упражнения, разработанные для обучения. Мы настоятельно рекомендуем не просто «читать» книгу, но «работать» с нею, т. е. выполнять все предлагаемые упражнения. Так вы усвоите гораздо больше информации и будете готовы встать на путь к созданию собственных проектов D3!

Для каждого упражнения и проекта мы подготовили файлы с исходным кодом. Вы можете найти их в репозитории книги на GitHub (http://mng.bz/Xqjv). Если вы знакомы с Git, то можете клонировать репозиторий на свой компьютер. Также можно скачать заархивированные файлы.



Загрузка файлов из репозитория GitHub

Для каждой главы в репозитории на GitHub создана отдельная папка, содержащая одно или несколько упражнений, пронумерованных по разделам главы. Упражнения включают папку `start`, содержащую все файлы, необходимые для начала. А полное решение вы найдете в папке `end`. По мере чтения разделов глав вы можете продолжать писать код в файле, который использовали в предыдущем разделе, или начать заново с папки, соответствующей текущему разделу. Оба варианта приведут к одному и тому же результату.

Итак, приступим к изучению векторной графики. Перейдите к файлам с кодом. Найдите папку `end` в `chapter_01/SVG_Shapes_Gallery` и щелкните правой кнопкой мыши на файле `index.html`. В контекстном меню выберите пункт **Open With** (Открыть с помощью) и затем выберите браузер. Мы рекомендуем выбирать Chrome или Firefox, потому что в них имеется замечательный набор инструментов разработчика. Файл откроется в новой вкладке браузера, и появится векторная графика, как показано на рис. 1.8. Эти фигуры SVG можно также рассмотреть в проекте, размещенном на GitHub (http://mng.bz/yZmB).



Рис. 1.8 Галерея основных фигур SVG, которые мы создадим в этом разделе

На рисунке изображены фигуры SVG, которые вы будете использовать чаще всего при создании визуализаций D3: прямые линии, прямоугольники, круги, эллипсы, кривые и текст.

При работе с D3 вы обычно сообщаете библиотеке, какие фигуры следует добавить в DOM. Вы также должны знать, какие атрибуты необходимо включить, чтобы фигуры имели нужные размеры, цвет и местоположение. В следующем упражнении мы предлагаем вам написать код, создающий каждый из элементов SVG, изображенных на рис. 1.8. Назовем это упражнение «Галерея фигур SVG». После этого вы будете знать все основы SVG, необходимые для начала работы.

Откройте файл `index.html` в папке `start` упражнения `SVG_Shapes_Gallery` в текстовом редакторе. Мы рекомендуем пользоваться Visual Studio Code (VS Code), бесплатным и удобным редактором кода со множеством функций, которые пригодятся вам при разработке интерфейса.

Как показано в листинге 1.1, `index.html` – это простой HTML-файл. Если открыть этот файл в браузере (щелкните правой кнопкой мыши на файле и в контекстном меню выберите пункт **Open With** (Открыть с помощью), затем выберите браузер), то вы увидите только пустую страницу, потому что элемент `<body>` пуст. В следующих подразделах мы добавим в него фигуры SVG.

Листинг 1.1 Начальный HTML-файл для упражнения «Галерея фигур SVG»

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Галерея фигур SVG | D3.js в действии</title>
</head>
<body>

</body>
</html>
```

Где найти более подробную информацию

В следующих разделах мы исследуем несколько элементов SVG и их атрибуты. Как разработчики интерфейсов мы ищем дополнительную информацию в интернете, когда используем незнакомые нам элементы SVG или выясняем, какую функцию JavaScript следует вызвать для выполнения определенного действия. Одним из самых надежных ресурсов в этом отношении мы считаем MDN Web Docs (https://developer.mozilla.org/). На этом сайте вы найдете простые и понятные примеры, демонстрирующие элементы SVG и их атрибуты, свойства CSS и функции JavaScript.

Адаптивный SVG-контейнер

В мире графики SVG-контейнер `<svg>` – это доска, на которой мы рисуем свою графику. Каждая отдельная фигура SVG вложена в родительский элемент `<svg>`. Чтобы увидеть его в действии, отредактируйте файл `index.html`, добавив контейнер SVG внутрь элемента `<body>`. Перезагрузив страницу в браузере, вы увидите, что она по-прежнему пустая:

```
<body>
  <svg></svg>
</body>
```

Откройте панель инспектора в браузере (щелкните правой кнопкой мыши в окне браузера и выберите **Inspect** (Исследовать)). В окне инспектора вы увидите дерево DOM страницы. Найдите контейнер `<svg></svg>`, также называемый узлом SVG. Если навести на него указатель мыши, то элемент SVG будет выделен на странице. Этот эффект показан на рис. 1.9.



Рис. 1.9 Узел SVG, выбранный в дереве DOM и выделенный в области просмотра

По умолчанию браузер задает ширину и высоту для контейнера SVG равными 300 и 150 пикселей соответственно. Но мы можем изменить эти значения, используя атрибуты. Атрибуты предназначены для передачи дополнительной информации об элементах HTML. В случае встроенной графики SVG в основном используются атрибуты, определяющие размеры и местоположение фигур.

Например, мы можем задать атрибуты ширины и высоты элемента SVG. Вернитесь в текстовый редактор и добавьте атрибуты `width` и `height` в контейнер SVG. Присвойте им значения `900` и `300` соответственно и сохраните файл:

```
<svg width="900" height="300"></svg>
```

Перезагрузите проект в браузере и найдите узел SVG в инспекторе. Обратите внимание, что теперь атрибуты ширины и высоты отображаются на элементе SVG. Если навести указатель мыши на узел SVG в дереве DOM в панели инспектора, то вы также увидите, что контейнер SVG в области просмотра теперь имеет размеры 900 на 300 пикселей, как показано на рис. 1.10.

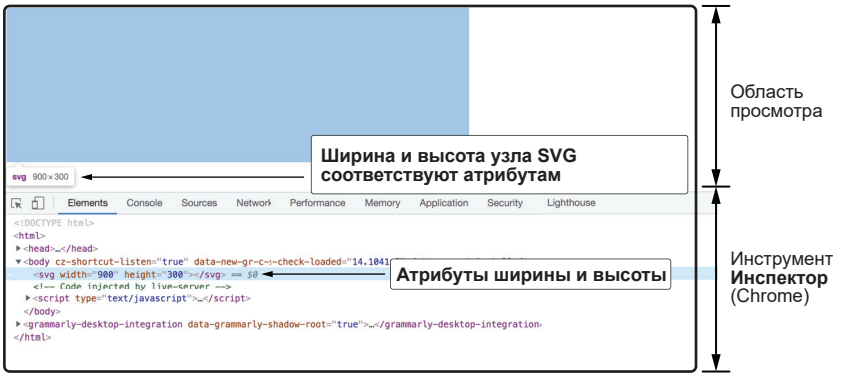


Рис. 1.10 Размеры узла SVG установлены согласно его атрибутам

Чтобы сделать контейнер SVG видимым без выбора его в инспекторе, добавим в него границы. Добавьте атрибут `style` в элемент SVG и присвойте ему CSS-свойство `border`. В следующем фрагменте мы использовали свойство `border`, определяющее черную сплошную границу толщиной в 1 пиксель:

```
<svg width="900" height="300" style="border:1px solid black;"></svg>
```

Сохраните файл, перезагрузите страницу и убедитесь, что вокруг контейнера SVG появилась граница. Теперь измените размер окна браузера, пока оно не станет меньше контейнера SVG. Вы увидите, что контейнер SVG сохраняет свою ширину и высоту и не подстраивается под размер окна браузера. Давайте попробуем сделать контейнер SVG адаптивным (автоматически подстраивающимся под размер окна).

Выше мы задавали атрибуты SVG как абсолютные значения (`900` и `300`), а браузер интерпретировал их как размеры в пикселях (`900 px` и `300 px`). Но мы также можем использовать проценты. В текстовом редакторе измените атрибут `width`, присвоив ему относительное значение `"100%"`, сохраните файл и перезагрузите страницу:

```
<svg width="100%" height="300" style="border:1px solid black;"></svg>
```

Снова измените размер окна браузера и обратите внимание, как SVG занимает всю доступную ширину, сохраняя фиксированную вы-

соту 300 пикселей. Уже лучше, но мы потеряли первоначальное соотношение сторон.

Для создания адаптивных элементов SVG можно использовать атрибут `viewBox`. В редакторе кода удалите атрибуты `width` и `height` из контейнера SVG и замените их атрибутом `viewBox`. Присвойте ему значение `"0 0 900 300"`:

```
<svg viewBox="0 0 900 300" style="border:1px solid black;"></svg>
```

Поиграйте еще раз с размерами окна браузера. Что вы заметили? Контейнер SVG теперь адаптируется к любому размеру экрана, сохраняя соотношение сторон 900:300. Мы получили адаптивный контейнер SVG!

Атрибут `viewBox`, как мы видели, принимает список из четырех значений. Первые два числа определяют начало системы координат в `viewBox` (`x` и `y`). В этой книге мы всегда будем использовать `0 0`, но знайте, что эти значения определяют видимую часть контейнера SVG. Последние два числа атрибута `viewBox` – это его ширина и высота. Они определяют соотношение сторон графики SVG и гарантируют ее идеальное масштабирование в любом контейнере без искажений.

Ключевым моментом здесь является размещение в контейнере. До сих пор роль контейнера для нашей графики SVG играл элемент `<body>`, который обычно расширяется и сжимается в соответствии с размерами области просмотра в окне браузера. Если размеры области просмотра увеличиваются, то увеличивается и графика SVG. Обычно бывает нужно, чтобы графика SVG имела максимальную ширину, но не становилась шире остального контента на странице. Для этого заключите контейнер SVG в `div` с шириной `100%` и максимальной шириной `1200 px`. Для простоты мы определили эти свойства как встроенные стили. Обратите также внимание, что мы добавили поле со значением `0 auto`, чтобы центрировать контейнер SVG по горизонтали:

```
<div style="width:100%; max-width:1200px; margin:0 auto;">
  <svg viewBox="0 0 900 300" style="border:1px solid black;"> ... </svg>
</div>
```

Попробуйте теперь изменить размер окна браузера и посмотрите, как SVG изящно адаптируется к любому размеру экрана, соблюдая ограничение максимальной ширины своего контейнера. Этот прием обязательно пригодится вам для внедрения визуализаций D3 в адаптивные веб-страницы, и мы часто будем использовать его на протяжении всей книги.

Система координат SVG

Теперь, узнав, как создать адаптивный встроенный контейнер SVG, займемся размещением фигур SVG внутри контейнера. Контейнер

SVG подобен чистому листу, на котором мы рисуем векторные фигуры. Векторные фигуры определяются с помощью основных геометрических правил и располагаются в соответствии с системой координат контейнера SVG.

Система координат SVG похожа на декартову систему координат. Ее 2-мерная плоскость имеет две перпендикулярные оси, определяющие положение элементов, которые называются x и y. Начало системы координат находится в верхнем левом углу контейнера SVG, как показано на рис. 1.11. Это означает, что положительное направление оси y простирается сверху вниз. Запомнив эту особенность, вы избавите себя от многих проблем!

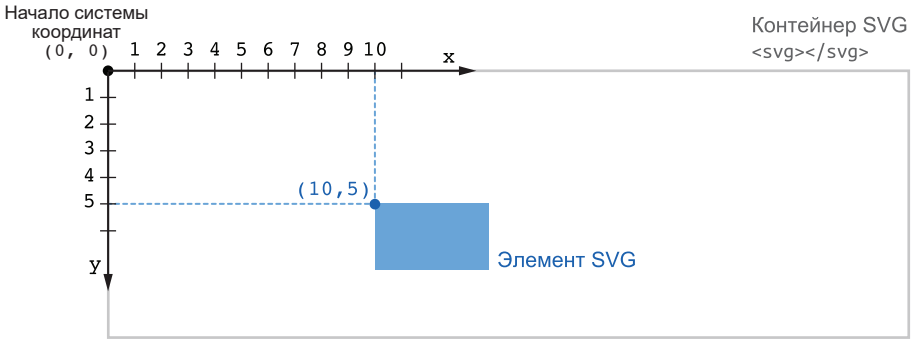


Рис. 1.11 Система координат контейнера SVG и местоположение элемента

При определении местоположения элемента внутри контейнера SVG его координата x откладывается вправо от начала координат в верхнем левом углу, и затем координата y откладывается вниз. Эти координаты местоположения определяются атрибутами представления каждой фигуры SVG.

Теперь рассмотрим формы SVG, которые часто используются в проектах D3. Мы также обсудим их основные атрибуты представления. Но имейте в виду, что мы не ставили перед собой цель написать всеобъемлющее руководство по всем формам и функциям SVG, а хотели охватить лишь базовые сведения, которые послужат вам основой при изучении D3.

Совет по визуализации данных: геометрические примитивы

С помощью векторной графики опытные художники могут нарисовать что угодно, но у вас, скорее всего, более прагматичный взгляд на D3. С этой точки зрения важно понимать суть геометрических примитивов (также известных как графические примитивы). *Геометрические примитивы* – это простые формы, такие как точки, отрезки, круги и прямоугольники. Эти формы, которые можно комбинировать для создания бо-

лее сложной графики, особенно удобны для визуального отображения информации.

Примитивы также полезны для понимания сложных визуализаций, которые встречаются в реальном мире. Древовидные макеты, подобные тем, которые мы построим в главе 11, будут выглядеть гораздо менее пугающими, когда вы поймете, что это всего лишь круги и отрезки. Интерактивные временные шкалы легче понять и создать, если думать о них как о наборах прямоугольников и точек. Даже географические данные, которые в основном представлены в виде многоугольников, точек и отрезков, кажутся менее запутанными, если разбить их на самые основные графические структуры.

Линия

Элемент `line` – самая простая из всех фигур SVG. Его местоположение определяется двумя точками, заданными как атрибуты, и изображается как отрезок прямой, соединяющий эти точки. Вернитесь к файлу `index.html` и добавьте элемент `<line />` внутри контейнера SVG. Объявите его атрибуты `x1` и `y1` со значениями `50` и `45` соответственно. Это означает, что начальная точка отрезка находится в точке `(50, 45)` в системе координат контейнера SVG. Начав с верхнего левого угла контейнера SVG и переместившись на 50 пикселей вправо и на 45 пикселей вниз, вы окажетесь в этой начальной точке. Аналогично задайте конечную точку `(140, 225)`, определив атрибуты `x2` и `y2`, как на рис. 1.12:

```
<svg>
    <line x1="50" y1="45" x2="140" y2="225" />
</svg>
```

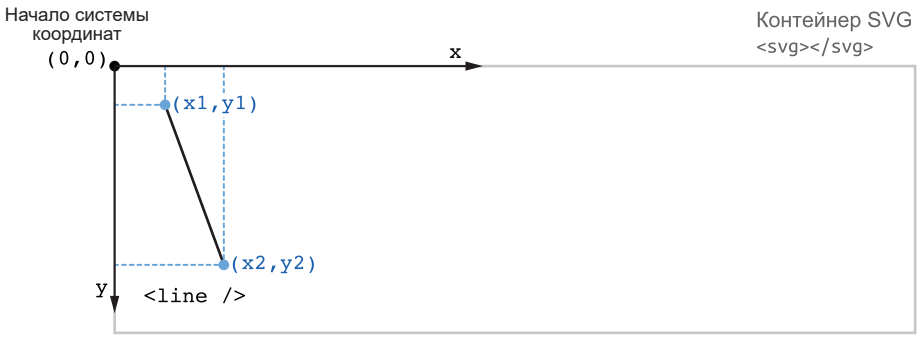


Рис. 1.12 Расположение элемента line в системе координат контейнера SVG

Если теперь сохранить и перезагрузить проект, то отрезок не появится на экране. В чем же дело? А дело в том, что нужно еще задать атрибут `stroke`, управляющий цветом линии. Значение атри-

бута `stroke` похоже на CSS-свойство `color`. Это может быть название цвета (`black`, `blue` и т. д.), цвет в формате RGB (`rgb(255,0,0)`) или шестнадцатеричное значение (`#808080`). Добавьте атрибут `stroke` в элемент `line` и присвойте ему цвет по своему выбору (мы задали черный цвет). Теперь линия должна быть видна на экране (рис. 1.12):

```
<line x1="50" y1="45" x2="140" y2="225" stroke="black" />
```

Задать определенную толщину линии можно с помощью атрибута `stroke-width`. Он принимает абсолютное число, которое преобразуется в пиксели, или относительное значение (%). Например, следующая линия будет иметь толщину 3 пикселя. Если атрибут `stroke-width` не объявлен, то браузер использует значение по умолчанию `1 px`:

```
<line x1="50" y1="45" x2="140" y2="225" stroke="black" stroke-width="3" />
```

Откройте панель инспектора в браузере и найдите узел SVG и содержащуюся в нем строку. Дважды щелкните на одном из атрибутов, измените его значение и посмотрите, как новое значение изменит начальную или конечную точку линии. Поэкспериментируйте с разными значениями, чтобы лучше понять, как атрибуты `x1`, `y1`, `x2` и `y2` влияют на местоположение и длину линии.

Теперь присвойте атрибуту `x1` значение `-20`. После этого начальная точка линии исчезнет, как показано на рис. 1.13. Любая форма или часть формы, которая оказывается *за пределами* `viewBox`, не отображается на экране. Однако элемент все еще существует в DOM, поэтому к нему можно обращаться и управлять им. Если элемент в контейнере SVG не виден, и вы не знаете почему, то прежде всего проверьте, не оказывается ли он за пределами `viewBox`! Помните, что вы всегда сможете отыскать элемент в дереве DOM с помощью инструментов разработчика. Как мы уже делали ранее, если навести указатель мыши на элемент в панели инспектора, то он будет выделен в области просмотра, даже если находится за пределами `viewBox`.

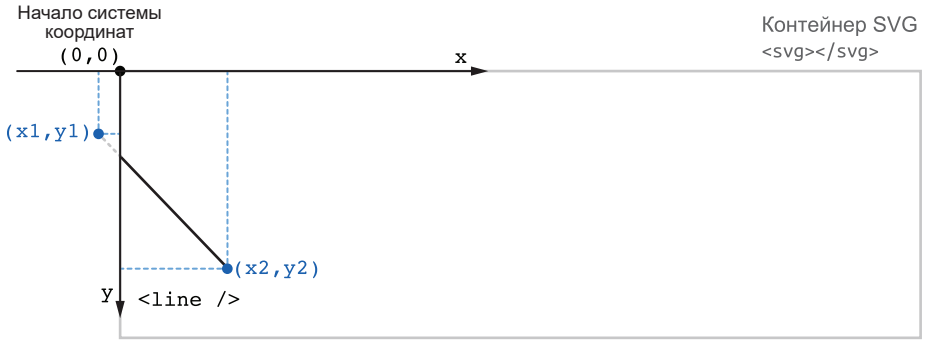


Рис. 1.13 Линия SVG частично скрыта, когда один из концов оказывается за пределами контейнера SVG

ПРИМЕЧАНИЕ Большинство элементов SVG можно определить с помощью единственного самозакрывающегося тега (в примерах выше мы использовали `<line />` вместо `<line></line>`). Как и некоторые другие теги HTML, внутренняя структура элементов SVG определяет всю необходимую информацию внутри самозакрывающегося тега. Но это не относится к текстовым элементам SVG, в которых текст размещается между открывающим и закрывающим тегами.

Прямоугольник

Элемент прямоугольника `<rect />` рисует на экране прямоугольную фигуру. В нем нужно определить четыре атрибута. Атрибуты `x` и `y` задают положение верхнего левого угла прямоугольника, а атрибуты `width` и `height` – ширину и высоту, как показано на рис. 1.14. Добавьте следующий элемент `<rect />` с атрибутами в контейнер SVG:

```
<rect x="260" y="25" width="120" height="60" fill="#6ba5d7" />
```

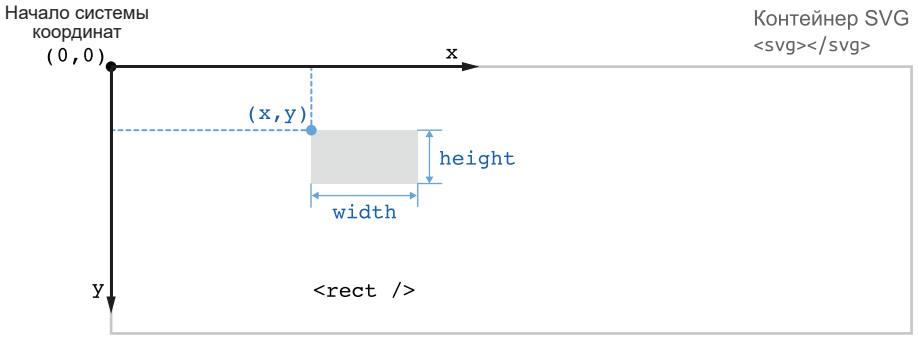


Рис. 1.14 Определение позиции и размеров прямоугольника в системе координат контейнера SVG

В нашем примере верхний левый угол прямоугольника расположен на 260 пикселей правее и на 25 пикселей ниже начала координат контейнера SVG. Он имеет ширину 120 пикселей и высоту 60 пикселей. По аналогии с другими позиционными атрибутами, их значения можно выразить в процентах. Например, если присвоить атрибуту `width` значение `50%`, то прямоугольник займет половину ширины контейнера SVG.

Возможно, вы заметили, что в первый момент прямоугольник окрасился в черный цвет. По умолчанию браузеры применяют черную заливку к большинству фигур SVG. Цвет заливки можно изменить, добавив атрибут `fill` и присвоив ему значение цвета в формате CSS. Чтобы нарисовать рамку вокруг прямоугольника, нужно добавить атрибут `stroke`. На рис. 1.15 показано несколько примеров

влияния атрибутов `fill` и `stroke`. Обратите внимание, что без атрибута `stroke` рамка вокруг прямоугольника не рисуется. Кроме того, в последнем прямоугольнике определены атрибуты `fill-opacity` и `stroke-opacity`, которые делают заливку и рамку полупрозрачными. Так же, как в CSS, непрозрачность можно задать как абсолютное значение (от 0 до 1) или в процентах (30 %). Все атрибуты, связанные с заливкой и рамкой, также можно изменить в файле CSS.

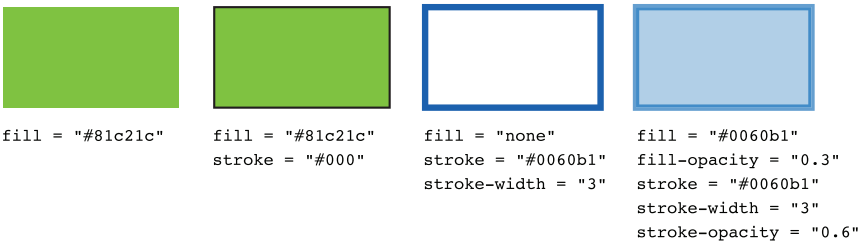


Рис. 1.15 Различные атрибуты стиля, применяемые к прямоугольным фигурам SVG

При желании углы прямоугольника можно закруглить, добавив атрибуты `rx` и `ry`, которые представляют горизонтальный и вертикальный радиус закругления угла соответственно. Эти атрибуты принимают абсолютные (в пикселях) и относительные значения (проценты). Например, каждый угол следующего прямоугольника имеет радиус закругления 20 пикселей. Добавьте этот прямоугольник в свою галерею фигур:

```
<rect x="260" y="100" width="120" height="60" rx="20" ry="20"
      fill="#6ba5d7" />
```

Теперь у вас может созреть вопрос: существует ли в SVG элемент для рисования квадратных фигур? Нет, но он и не нужен! Квадраты в SVG рисуются как элементы `<rect />` с равными значениями атрибутов `width` и `height`. Например, следующий элемент `<rect />` нарисует квадрат размером 60×60 пикселей. Добавьте его в свою галерею фигур:

```
<rect x="260" y="175" width="60" height="60" fill="transparent"
      stroke="#6ba5d7" />
```

Теперь в нашей галерее фигур есть три типа прямоугольников: классический прямоугольник, прямоугольник с закругленными углами и квадрат. Ради забавы мы присвоили им цвет `#6ba5d7` и поиграли с их атрибутами `stroke` и `fill`. Обратите внимание, что от квадрата видна только рамка, потому что его атрибут `fill` имеет значение `transparent` («прозрачный», тот же эффект даст значение `none`). У вас прямоугольники должны выглядеть примерно так же,

как на рис. 1.16, если только вы не использовали другие значения в атрибутах, что мы рекомендуем вам сделать:

```
<rect x="260" y="25" width="120" height="60" fill="#6ba5d7" />
<rect x="260" y="100" width="120" height="60" rx="20" ry="20"
➥ fill="#6ba5d7" />
<rect x="260" y="175" width="60" height="60" fill="transparent"
➥ stroke="#6ba5d7" />
```

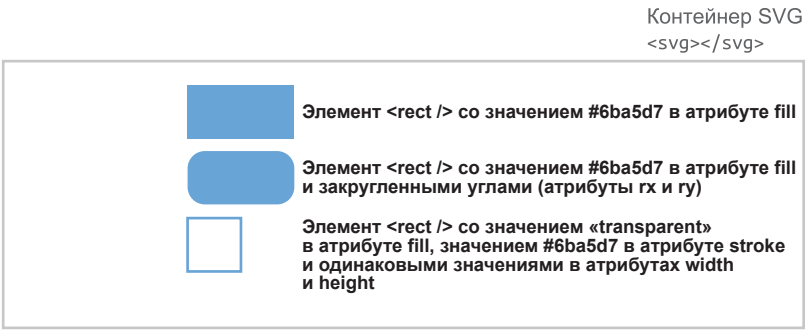
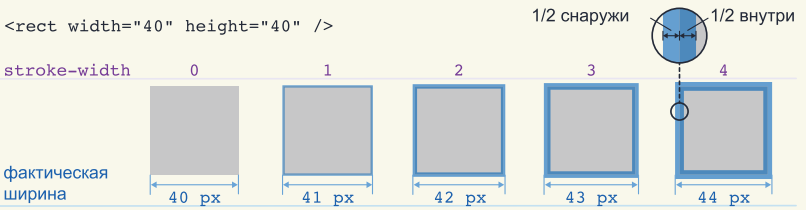


Рис. 1.16 Три типа прямоугольников SVG

Местоположение рамок в фигурах SVG

Пытаясь выравнивать фигуры в визуализации, помните, что по умолчанию рамки наполовину рисуются по внутренней границе, а наполовину по внешней. Как показано на следующем рисунке, применение `stroke-width` со значением `1` к прямоугольнику с атрибутом `width`, равным `40` px, визуально добавит 0,5 пикселя слева и 0,5 пикселя справа от прямоугольника (а не по 1 пикселю с каждой стороны, как можно было бы подумать), что даст фактическую общую ширину в 41 пиксель. Применение атрибута `stroke-width` со значением `2` добавит по 1 пикселю с каждой стороны и т. д.



Влияние stroke-width на фактическую ширину фигуры SVG

Круг и эллипс

Круглые формы регулярно используются в визуализации данных. Они естественным образом привлекают взгляд и делают визуали-

зацию более дружелюбной и привлекательной. Круги в SVG рисуются с помощью элемента `<circle />`. Его обязательными атрибутами являются координаты центра (`cx`, `cy`) и радиус (`r`), как показано на рис. 1.17. Радиус круга – это длина линии, проведенной от центра до любой точки на его окружности. Добавьте следующий круг в свою галерею фигур. Поместите его центр в точку с координатами `(530, 80)` и задайте радиус равным 50 пикселям:

```
<circle cx="530" cy="80" r="50" />
```

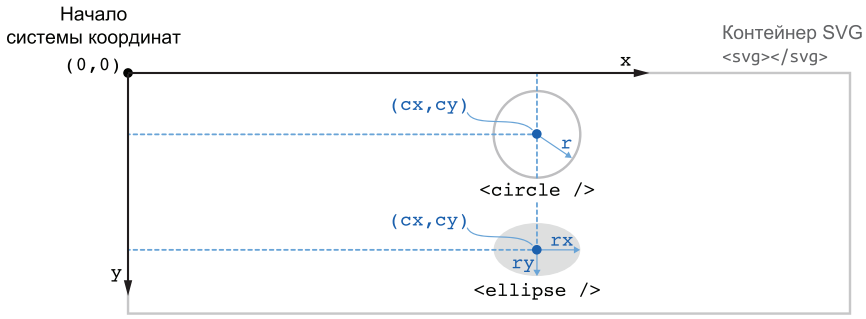


Рис. 1.17 Местоположение и размер круга и эллипса в системе координат контейнера SVG

Вы можете дополнительно поэкспериментировать с атрибутами `fill` и `stroke` круга. В своей галерее (рис. 1.17) мы использовали прозрачную заливку и рамку толщиной 3 пикселя с цветом `#81c21c`.

Элемент `<ellipse />` тоже требует определить атрибуты, задающие центр фигуры (`cx`, `cy`). Но в отличие от кругов, имеющих постоянный радиус, в эллипсах радиус меняется, что придает им сплющенную форму. Эффект сплющивания задается с помощью атрибутов горизонтального и вертикального радиусов – `rx` и `ry` соответственно. Добавьте следующий фрагмент в свою галерею. Он нарисует эллипс под кругом с горизонтальным радиусом 50 пикселей и вертикальным радиусом 30 пикселей:

```
<ellipse cx="530" cy="205" rx="50" ry="30" />
```

Кривая

Кривые SVG являются, безусловно, самыми гибкими из всех элементов. Они широко используются в D3 для рисования практически всех сложных форм, которые нельзя представить ни одним из примитивов, обсуждавшихся выше (линия, прямоугольник, круг и эллипс).

Мы указываем браузеру, как рисовать кривую, объявляя атрибут `d` (от слова «draw» – «рисовать»). Атрибут `d` содержит список команд, от начальной координаты рисования до типов используемых кривых, вплоть до того, что мы можем потребовать сделать кривую замкну-

той. В качестве примера добавьте следующий элемент в свою галерею фигур:

```
<path d="M680 150 C 710 80, 725 80, 755 150 S 810 220, 840 150" fill="none"
      stroke="#773b9a" stroke-width="3" />
```

В этом примере (рис. 1.18) атрибут `d` начинается с `M680 150`, что означает «move (перейти) в точку с координатами (680, 150)». Затем запускается рисование кубической кривой Безье от начальной точки (680 150) до конечной точки, указанной последней парой чисел (840 150) в атрибуте `d`. Для отображения кубической кривой Безье также нужно задать контрольные точки, определяющие крутизну и направления изгибов кривой. Определение контрольных точек начинается с символа C («cubic curve») и заканчивается сразу после буквы S («stop»).

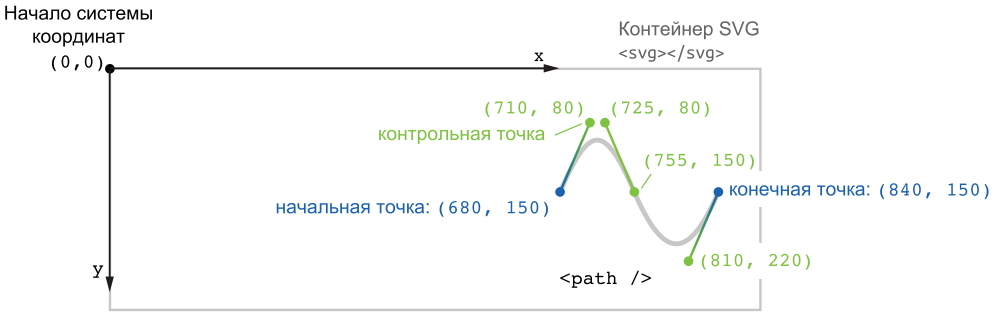


Рис. 1.18 Простая кривая SVG и ее контрольные точки

ПРИМЕЧАНИЕ Более полную информацию о кривых SVG вы найдете в руководстве MDN по адресу https://mng.bz/amn7.

Для простых кривых можно вручную писать значения для атрибута `d`, но по мере их усложнения эта задача становится весьма утомительной. К счастью, D3 предлагает мощные генераторы форм, способные автоматически вычислять значения для атрибута `d`. Мы познакомимся с ними в главе 4.

Также важно помнить, что браузеры заливают кривые черным цветом, если не присвоить атрибуту `fill` значение `none` или `transparent`. Это верно даже в том случае, если кривая не замкнута, как в нашем примере.

Текст

Одно из самых значительных преимуществ встроенной графики SVG – доступность текста, содержащегося в ней, для устройств чтения с экрана. Это большой плюс для людей с ограниченными возмож-

ностями. Визуализации часто содержат метки и надписи, поэтому важно знать, как манипулировать текстом SVG с помощью элемента `<text>`. Давайте добавим метки в нашу галерею фигур, чтобы понять основные принципы оформления текста SVG.

Обсуждавшиеся до сих пор фигуры SVG определялись с помощью самозакрывающихся тегов (`<line />`, `<rect />`, `<path />` и т. д.). При работе с элементами `<text>` нужно использовать оба тега, открывающий и закрывающий. Отображаемый текст помещается между этими двумя тегами. Например, давайте добавим элемент `<text>` со словами «прямая линия»:

```
<text>прямая линия</text>
```

Сохраните файл и перезагрузите страницу. Возможно, вы ожидали, что текст появится в верхнем левом углу контейнера SVG, но его нигде не видно. Почему? По умолчанию положение текста SVG вычисляется относительно его базовой линии, контролируемой атрибутом `dominant-baseline`. Если координата базовой линии текста равна `(0,0)`, то, как показано на рис. 1.19, фактический текст оказывается за пределами контейнера SVG. Любой элемент, оказавшийся за пределами контейнера SVG, невидим, поэтому мы не видим наш текст.

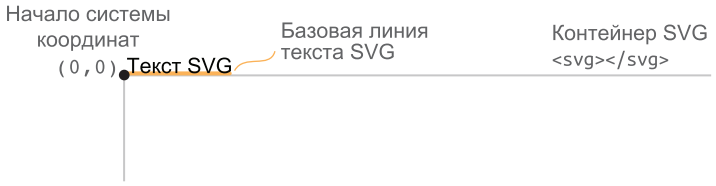


Рис. 1.19 По умолчанию текст располагается за пределами контейнера SVG

Еще один аспект, который следует учитывать при работе с текстом SVG, – его перенос. Обычные элементы HTML располагаются на странице в соответствии с определенными правилами, которые управляют потоком содержимого. Если добавить в страницу несколько элементов `<div>`, они будут накладываться друг на друга, а текст в них будет переноситься на строки так, чтобы он не вышел за пределы своего контейнера. Текст SVG не переносится, поэтому необходимо задавать атрибуты `x` и `y` для каждого текстового элемента отдельно. Если использовать эти атрибуты для размещения текста в точке `(60,260)`, то метка «прямая линия» появится в нашей галерее фигур под линией SVG:

```
<text x="60" y="260">прямая линия</text>
```

Для практики создайте новый текстовый элемент, помещающий подпись «прямоугольник» под прямоугольником и квадратом.

До сих пор мы использовали атрибуты x и y для объявления положения нижнего левого угла наших текстовых элементов. А как быть, если понадобится задать местоположение средней точки текста? Это можно сделать с помощью атрибута `text-anchor` со значением `middle`, как показано на рис. 1.20. Применив этот атрибут, можно, например, отцентрировать текстовую подпись под кругом:

```
<text x="530" y="155" text-anchor="middle">circle</text>
```

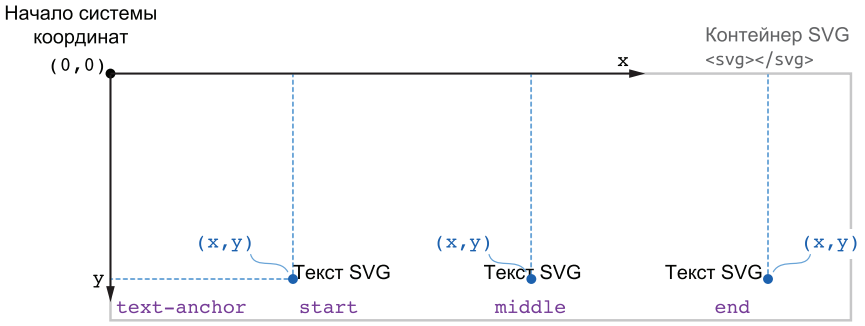


Рис. 1.20 Атрибут text-anchor влияет на выравнивание текста SVG. По умолчанию он получает значение start. Чтобы выровнять текстовый элемент по середине, нужно применить атрибут `text-anchor="middle"`. Аналогично, чтобы выровнять текст по концу, нужно применить атрибут `text-anchor="end"`

В заключение добавьте подпись под эллипсом и под кривой. По умолчанию текст SVG имеет черный цвет, но его можно поменять с помощью атрибута `fill`.

Группировка элементов

Последним в этом разделе мы обсудим группирующий элемент SVG. Группирующий элемент `<g>` отличается от элементов SVG, обсуждавшихся до сих пор, тем, что он не имеет графического представления и не определяет ограниченного пространства, а просто логически объединяет несколько элементов. Вам наверняка придется использовать группы при создании визуализаций, состоящих из нескольких фигур и меток.

Если нужно, чтобы квадрат и метка «прямоугольник» отображались вместе и перемещались как одно целое в контейнере SVG, их можно поместить внутрь элемента `<g>`, как в следующем примере. Обратите внимание, что теперь координаты верхнего левого угла элемента `<rect>` установлены равными `(0, 0)`. Элемент `<text>` помещен в точку с координатами `(0, 85)` ниже элемента `<rect>`:

```
<g>
  <rect x="0" y="0" width="60" height="60" />
  <text x="0" y="85">прямоугольник</text>
</g>
```

Группа, содержащая квадрат и его метку, теперь отображается в левом верхнем углу контейнера SVG. Группу и все ее элементы можно произвольно перемещать в пределах контейнера SVG, сохраняя при этом выравнивание между квадратом и его меткой.

Перемещение группы выполняется с помощью атрибута `transform`. Он выглядит не так, как другие атрибуты, обсуждавшиеся до сих пор, но в целом идентичен CSS-свойству `transform`. Он принимает преобразование (перенос, поворот, масштабирование и т. д.) или стек преобразований. Чтобы переместить группу, мы использовали преобразование `translate(x, y)`, указав, что группа с элементами `<rect>` и `<text>` должна быть смещена на 260 пикселей вправо и на 175 пикселей вниз:

```
<g transform="translate(260,175)">
  <rect x="0" y="0" width="60" height="60" />
  <text x="0" y="85">rect</text>
</g>
```

Другое полезное свойство элемента `<g>` – его потомки наследуют от него атрибуты стиля. Для иллюстрации давайте сгруппируем все оставшиеся текстовые элементы внутри элемента `<g>`, за исключением метки «прямоугольник», которую мы уже сгруппировали с квадратом:

```
<g>
  <text x="60" y="260">line</text>
  <text x="530" y="155" style="text-anchor:middle">circle</text>
  <text x="530" y="260" style="text-anchor:middle">ellipse</text>
  <text x="730" y="260">path</text>
</g>
```

Если к группе применить атрибут `fill="#636466"`, то каждый элемент `<text>` внутри этой группы унаследует тот же цвет. Аналогично если применить атрибут `style`, например, со свойствами `font-family` и `font-size`, то текст внутри группы унаследует и эти свойства:

```
<g fill="#636466" style="font-size:16px; font-family:monospace">
  <text x="60" y="260">line</text>
  <text x="530" y="155" style="text-anchor:middle">circle</text>
  <text x="530" y="260" style="text-anchor:middle">ellipse</text>
  <text x="730" y="260">path</text>
</g>
```

Перезагрузите страницу еще раз и убедитесь, что метки внутри группы унаследовали цвет и шрифт группы, в то время как метка «прямоугольник», объявленная вне этой группы, сохраняет свой первоначальный вид. Этот метод применения общих атрибутов

к элементам группы очень удобен. Он может помочь следовать принципу программирования «не повторяйся» (Don't Repeat Yourself, DRY), а также облегчит вам жизнь, когда понадобится изменить эти атрибуты.

А теперь примите наши поздравления с успешным завершением первого упражнения! Полный код галереи фигур SVG вы найдете в листинге 1.2 и в папке end. Используйте это упражнение в качестве справки при создании своих первых проектов D3.

Листинг 1.2 Полный код проекта «Галерея фигур SVG»

```
<!DOCTYPE html>
<html>
<head> [...] </head>
<body>
  <div style="width:100%; max-width:1200px; margin:0 auto;">
    <svg viewBox="0 0 900 300" style="border:1px solid black;">

      <line x1="50" y1="45" x2="140" y2="225" stroke="black" />

      <rect x="260" y="25" width="120" height="60" fill="#6ba5d7" />
      <rect x="260" y="100" width="120" height="60" rx="20" ry="20"
            fill="#6ba5d7" />
      <g transform="translate(260, 175)">
        <rect x="0" y="0" width="60" height="60" fill="transparent"
              stroke="#6ba5d7" />
        <text x="0" y="85">прямоугольник</text>
      </g>

      <circle cx="530" cy="80" r="50" fill="none" stroke="#81c21c"
              stroke-width="3" />
      <ellipse cx="530" cy="205" rx="50" ry="30" fill="#81c21c" />

      <path d="M680 150 C 710 80, 725 80, 755 150 S 810 220, 840 150"
            fill="none" stroke="#773b9a" stroke-width="3" />

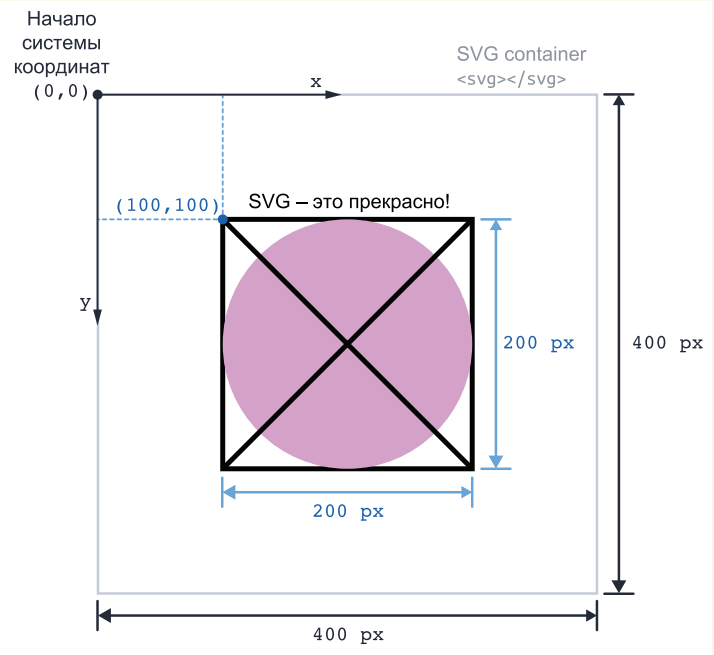
      <g fill="#636466" style="font-size:16px; font-family:monospace">
        <text x="60" y="260">прямая линия</text>
        <text x="530" y="155" style="text-anchor:middle">круг</text>
        <text x="530" y="260" style="text-anchor:middle">эллипс</text>
        <text x="730" y="260">кривая</text>
      </g>

    </svg>
  </div>
</body>
</html>
```

Упражнение: создание SVG-графики

Теперь ваш черед! Создайте графику SVG, показанную на следующем рисунке. Вы можете начать с файлов в папке `start` внутри каталога `02_SVG_exercise`. Вот что вы должны сделать:

- создайте адаптивный контейнер SVG с максимальной шириной и высотой 400 пикселей (когда на экране достаточно места);
- нарисуйте квадрат шириной и высотой 200 пикселей в центре контейнера SVG, задайте ему прозрачную заливку и черную рамку толщиной 5 пикселей;
- добавьте круг радиусом 100 пикселей в центре контейнера SVG, присвойте его атрибуту `fill` CSS-цвет «plum»;
- нарисуйте две диагональные черные линии с рамкой толщиной в 5 пикселей. Одна должна соединять верхний левый и правый нижний углы квадрата, а другая – правый верхний и левый нижний;
- добавьте текст «SVG – это прекрасно!» над квадратом и отцентрируйте его по горизонтали в контейнере SVG. Придайте тексту следующие свойства стиля: размер шрифта (`font-size`) 18 пикселей и семейство шрифтов (`font-family`) sans-serif.



Обязательно создайте эту SVG-графику, чтобы закрепить идеи, обсуждавшиеся в данном разделе

Решение вы найдете в разделе D.1.1 приложения D и в папке `02_SVG_exercise/end` в репозитории с кодом для этой главы. Постарайтесь выполнить данное упражнение самостоятельно.

1.2.3 Canvas и WebGL

Обычно мы создаем проекты D3, используя элементы SVG, но иногда бывает нужно создать сложные визуализации из больших наборов данных, когда традиционный подход SVG может породить проблемы с производительностью. Важно помнить, что D3 добавляет один или несколько элементов SVG в DOM для каждой графической детали в визуализации данных. Типичным примером является визуализация большой сети, состоящей из тысяч узлов и ссылок. Отображение такой визуализации может вызвать большую нагрузку на ваш браузер. Количество объектов, которые браузеры могут довольно быстро обрабатывать, постоянно увеличивается, тем не менее бытует общепринятое практическое правило – подумать о возможности использования Canvas вместо SVG, если визуализация насчитывает более 1000 элементов.

Canvas – это API для рисования на стороне клиента с помощью набора инструкций, обычно на JavaScript. Он не добавляет элементы XML в DOM, что значительно повышает производительность при построении визуализаций из больших наборов данных. Canvas не дает такой же четкой визуализации, как SVG, и добавляет дополнительные сложности, поэтому мы обычно стараемся ограничиться средствами SVG.

Canvas также позволяет использовать Web Graphics Library (WebGL) API для создания 3-мерных объектов. Мы не будем изучать WebGL в этой книге, но вы должны знать, что создание 3-мерных визуализаций данных на веб-страницах вполне возможно. В настоящее время Canvas в основном используется в экспериментальных проектах. В главе 15 мы рассмотрим пример создания визуализации с помощью Canvas и обсудим плюсы и минусы этого средства.

1.2.4 CSS

CSS (что означает *Cascading Style Sheets* – каскадные таблицы стилей) – это язык, описывающий способы представления элементов DOM на экране и их внешний вид. От общей сетки макета страницы до семейства шрифтов и цвета кругов на диаграмме рассеяния CSS может превратить простой HTML-файл во впечатляющую веб-страницу. В проектах D3 мы обычно используем встроенные стили CSS или применяем оформление, описанное во внешней таблице стилей.

Встроенные стили применяются к элементам с помощью атрибута `style`. Атрибут `style` можно добавлять и в традиционные элементы HTML, и в элементы SVG, а кроме того, D3 предлагает удобный метод установки или изменения этого атрибута, который мы обсудим в главе 2. Следующий пример демонстрирует атрибут `style` со встроенными стилями:

```
<div style="padding:10px; background:#00ced1;"> ... </div>
<text style="font-size:16px; font-family:serif;"> ... </text>
```

Встроенные стили влияют только на элемент, к которому они применяются. Чтобы распространить одно и то же оформление на несколько элементов, нужно применить один и тот же атрибут `style` к каждому из них (или к группе SVG, которая обертывает несколько элементов). Это вполне действенный, но очень неэффективный способ.

Внешние таблицы стилей CSS, напротив, идеально подходят для глобального применения стилей. Стратегия заключается в том, чтобы потребовать от D3 добавить одно и то же имя класса в несколько элементов, а затем использовать это имя класса как селектор во внешней таблице стилей для применения оформления к целевой группе элементов, как показано в следующей таблице стилей:

```
.my-class {
    font-size: 16px;
    font-family: serif;
}
```

и в DOM:

```
<text class="my-class"> ... </text>
```

Этот подход гораздо эффективнее, особенно в больших проектах. Он также следует принципу разделения ответственности, помогая отделить поведение, определяемое кодом JavaScript, от оформления, определяемого стилями CSS. Обратите внимание, что препроцессоры CSS, такие как Syntactically Awesome Stylesheets (SASS) и Leaner Stylesheets (LESS), являются частью подхода на основе внешних таблиц стилей, описанного здесь.

> **СОВЕТ** Помните, что встроенные стили имеют приоритет перед стилями, применяемыми из внешней таблицы стилей. В любом проекте важно заранее спланировать архитектуру стилей CSS с учетом каскадного порядка.

1.2.5 JavaScript

D3 – это библиотека JavaScript, которая добавляет новые методы к существующим функциям JavaScript. Это означает, что даже небольшой опыт работы с JavaScript пригодится при работе с D3. Кроме того, в проектах D3 у вас сохраняется доступ ко всем существующим функциям JavaScript.

Полноценное знакомство с JavaScript может занять целую книгу, поэтому в данном разделе мы рассмотрим только две темы, связан-

ные с JavaScript, которые широко используются в проектах D3: *объединение вызовов методов в цепочки* и *манипулирование объектами*.

Объединение вызовов методов в цепочки

Поискав примеры проектов D3 в интернете, вы заметите, что в них методы часто вызываются один за другим. Мы называем это объединением вызовов методов в цепочки. Такой прием помогает сохранить код лаконичным и читабельным.

Цепочку методов можно рассматривать как сборочный конвейер. Допустим, мы пишем сценарий, который запускает такой конвейер. Как показано в следующем примере, сначала объявляется переменная `car`, которая создает новый объект `Car()`. Затем вызывается функция `putOnHood()`, прикручивающая капот к автомобилю, и продолжаем вызывать функции, добавляющие колеса, шины и фары. Каждый следующий вызов добавляет очередную деталь в объект `Car()`, и в результате мы получаем автомобиль с капотом, колесами, шинами и фарами. Каждый метод передает обновленный объект автомобиля следующему, так и выстраивается «цепочка». Обратите внимание, что каждый вызов отделен от предыдущего точкой, и методы выполняются в том порядке, в каком они перечислены в цепочке. В нашем примере сборочного конвейера мы должны установить колеса прежде, чем надеть на них шины. Пример выглядит так:

```
let car = new Car().putOnHood().putOnWheels().putOnTires().putOnLights();
```

Теперь посмотрим, как мы будем использовать цепочки вызовов методов в D3. Представьте, что мы хотим извлечь все элементы `<div>` из DOM и в каждый добавить элемент абзаца. Элементы абзаца должны иметь атрибут `class="my-class"` и содержать текст «Класс!». Затем в каждый абзац вставить элемент `<span>` с текстом «Вообще улет!» жирным шрифтом. Без использования цепочек методов нам пришлось бы сохранять каждое действие в константу, а затем вызывать эту константу для выполнения следующего действия, как показано здесь (на этот код утомительно даже смотреть, не то что писать его):

```
const mySelection = d3.selectAll("div");
const myParagraphs = mySelection.append("p");
const myParagraphsWithAClass = myParagraphs.attr("class", "my-class");
const myParagraphsWithText = myParagraphsWithAClass.text("Класс!");
const mySpans = myParagraphsWithText.append("span");
const mySpansWithText = mySpans.text("Вообще улет!")
const myBoldSpans = mySpansWithText.style("font-weight", "900");
```

Но благодаря возможности составления цепочек из вызовов методов то же самое можно записать более кратко:

```
d3.selectAll("div").append("p").attr("class", "my-class").
➥ text("Класс!").append("span").text("Вообще улет!").
➥ style("font-weight", "900");
```

При оформлении кода D3 принято разбивать строки, к чему JavaScript относится благосклонно, и оформлять одинаковые отступы в группах взаимосвязанных методов. Это облегчает чтение кода, а отступы помогают увидеть, с каким элементом выполняются манипуляции:

```
d3.selectAll("div")
  .append("p")
    .attr("class", "my-class")
    .text("Класс!")
  .append("span")
    .text("Вообще улет!")
    .style("font-weight", "900");
```

Не волнуйтесь, если пока не до конца понимаете, как работает предыдущий код. Сейчас мы лишь знакомимся с возможностью объединения вызовов методов в цепочки. В главе 2 мы дополнительно познакомимся с жаргоном, специфичным для D3.

Манипулирование массивами и объектами

D3 – это библиотека для работы с данными, а данные часто структурированы как объекты JavaScript. Понимание устройства этих объектов и способов получения доступа к данным, которые они содержат, а также приемов манипулирования ими здорово помогает при построении визуализаций.

Давайте для начала поговорим о простых массивах – обычных списках элементов. В проектах, связанных с данными, массивы часто представляют собой упорядоченный список чисел или строк:

```
const arrayOfNumbers = [17, 82, 9, 500, 40];
const arrayOfStrings = ["blue", "red", "yellow", "orange"];
```

Каждый элемент массива имеет числовую позицию, называемую *индексом*, а первый элемент имеет индекс `0` (т. е. нумерация элементов в массивах JavaScript начинается с нуля):

```
arrayOfNumbers[0]   // => 17
arrayOfStrings[2]   // => "yellow"
```

Массивы имеют свойство `length`, которое для неразреженных массивов равно количеству содержащихся в них элементов. Поскольку нумерация элементов в массивах начинается с нуля, последний элемент в массиве имеет индекс, соответствующий длине массива минус 1:

```
arrayOfNumbers.length;                    // => 5
arrayOfStrings[arrayOfStrings.length - 1]   // => "orange"
```

Также с помощью метода `includes()` можно определить присутствие в массиве определенного значения. Этот метод возвращает `true`, если в массиве есть хотя бы один элемент, значение которого точно соответствует значению аргумента; в противном случае он возвращает `false`:

```
arrayOfNumbers.includes(9)         // => true
arrayOfStrings.includes("pink")    // => false
arrayOfStrings.includes("ellow")   // => false
```

Однако большинство наборов данных – не просто списки чисел или строк, а содержат точки данных, каждая из которых состоит из нескольких свойств. Давайте представим базу данных сотрудников вымышленного агентства, представленную в табл. 1.1. Таблица содержит четыре столбца: идентификатор (`Id`), имя (`name`) и должность (`Position`) каждого сотрудника, а также признак владения приемами работы с D3 этим сотрудником (`Works_with_d3`).

Таблица 1.1 Небольшой набор данных с сотрудниками и их должностями

Id	Name	Position	Works_with_d3
1	Зоя	Аналитик данных	False
2	Джеймс	Разработчик веб-интерфейсов	True
3	Алиса	Разработчик полного профиля	True
4	Хуберт	Дизайнер	False

Каждую строку (или точку данных) в этом наборе можно представить объектом JavaScript, например `row1`:

```
const row1 = {
               id:"1",
               name:"Зоя",
               position:"Аналитик данных",
               works_with_d3:false
             };
```

Получить значение каждого свойства объекта можно с помощью точечной нотации:

```
row1.name            // => "Зоя"
row1.works_with_d3   // => false
```

или с помощью квадратных скобок. Скобочная нотация удобна, когда имя свойства содержит специальные символы, такие как пробелы, или если имя свойства хранится в константе либо переменной:

```
row1["position"]                    // => "Аналитик данных"

const myProperty = "works_with_d3";
row1[myProperty]                    // => false
```

В реальной жизни наборы данных обычно форматируются как массивы объектов. Например, если загрузить набор данных, показанный в табл. 1.1, с помощью D3, как вы научитесь делать это в главе 3, то получится следующий массив объектов, который можно сохранить в константе с именем `data`:

```
const data = [
 {id:"1", name:"Зоя", position:"Аналитик данных", works_with_d3:false},
 {id:"2", name:"Джеймс", position:"Разработчик веб-интерфейсов", works_with_d3:true},
 {id:"3", name:"Алиса", position:"Разработчик полного профиля",works_with_d3:true},
 {id:"4", name:"Хуберт", position:"Дизайнер", works_with_d3:false}
];
```

Мы можем выполнить обход элементов в наборе `data` с помощью цикла, в частности с помощью цикла `forEach` как наиболее удобного и простого. В процессе итерации часто выполняется некоторая обработка данных. Например, загружая внешний CSV-файл, мы можем форматировать числа как строки. Давайте возьмем наш массив данных и преобразуем строковые значения свойства `id` в числовые.

В следующем примере итератор `d` массива позволяет получить доступ к каждому элементу данных. Используя точечную нотацию, мы преобразуем каждый идентификатор в число с помощью оператора `+`:

```
data.forEach(d => {
  d.id = +d.id;
});
```

В JavaScript имеется множество методов обхода элементов массива, которые помогают взаимодействовать с данными и даже изменять их форму при необходимости. Допустим, мы решили визуализировать набор данных с сотрудниками. В таком случае может пригодиться простой массив, содержащий только имена сотрудников, и создать его можно с помощью метода `map()`:

```
data.map(d => d.name);   // => ["Зоя", "Джеймс", "Алиса", "Хуберт"]
```

Аналогично мы можем выбрать в массив только сотрудников, умеющих работать с D3, используя метод `filter()`:

```
data.filter(d => d.works_with_d3);

// => [
   {id:2, name:"Джеймс", position:"Разработчик веб-интерфейсов", works_with_d3:true},
   {id:4, name:"Алиса", position:"Разработчик полного профиля", works_with_d3:true}
  ];
```

Наконец, с помощью метода `find()` мы можем найти сотрудника с идентификатором `3`. Обратите внимание, что метод `find()` прекращает поиск, встретив первое подходящее значение. Его можно использовать только для поиска одной уникальной точки данных:

```
data.find(d => d.id === 3);


// => {id:"3", name:"Алиса", position:"Разработчик полного профиля",
works_with_d3:true}
```

В этом разделе мы обсудили далеко не все методы манипулирования массивами и объектами, имеющиеся в арсенале JavaScript, но именно к ним, скорее всего, вы будете возвращаться при работе с данными. Если вам понадобится другой способ доступа к данным или манипулирования ими, обращайтесь к MDN Web Docs (https://developer.mozilla.org/) – надежному справочнику с большим количеством примеров.

1.2.6 Фреймворки Node и JavaScript

JavaScript претерпел значительные изменения за последнее десятилетие. Две наиболее значимые тенденции в современном JavaScript – это развитие node.js и выбор фреймворков JavaScript на роль стандарта в большинстве проектов.

Основная технология node, которая широко используется в проектах D3, – это Node Package Manager (NPM). NPM позволяет устанавливать «модули» или небольшие библиотеки JavaScript. Вам не нужно включать кучу тегов `<script>` со ссылками на отдельные файлы, и часто модули не являются одной монолитной структурой, что помогает сократить объем кода, загружаемого приложением.

Версия 7 библиотеки D3.js, вышедшая в середине 2021 года, использует преимущества, которые дает импорт модулей. В этой книге вы увидите примеры использования D3 одним из двух способов: с загрузкой всей библиотеки D3 (в главе 2) и с включением лишь нужных частей D3 (в последующих примерах). Это можно сделать с помощью тегов `<script>`, но начиная с раздела 2 мы будем импортировать модули D3 с помощью NPM, поскольку в настоящее время это считается стандартной практикой. Вам, вероятно, понадобится поближе познакомиться с ней, если вы решите начать создавать проекты D3 на профессиональном уровне.

Если вы уже участвуете в профессиональных веб-проектах, то велика вероятность, что вы имеете опыт использования фреймворков JavaScript, таких как React, Angular, Vue или Svelte. Фреймворки дают разработчикам основу для создания веб-проектов с модульным, повторно используемым и тестируемым кодом. Они отвечают за соз-

дание и обновление DOM, что делает и библиотека D3. В главе 8 мы обсудим стратегии, позволяющие избежать конфликтов с фреймворками JavaScript при создании визуализаций D3.

Наконец, в профессиональной рабочей среде D3 можно использовать в сочетании с TypeScript. TypeScript – это синтаксическое надмножество JavaScript, улучшающее масштабируемость проекта и удобство поддержки кода. Хотя мы не будем подробно обсуждать это в данной книге, но вы можете установить типы методов D3 с помощью NPM-пакета @types/d3 (www.npmjs.com/package/@types/d3). В главе 8 мы будем использовать такие типы в проекте Angular.

1.2.7 Блокноты Observable

В поисках примеров проектов D3 в интернете вы обязательно столкнетесь с блокнотами Observable (https://observablehq.com). Observable – это среда для совместной работы в области науки о данных и визуализации, похожая на Jupyter Notebook для Python. Платформа Observable была создана Майком Бостоком (Mike Bostock) и заменила *blocks* – предыдущую онлайн-песочницу D3. Все официальные примеры D3 теперь находятся на Observable, и там вы найдете довольно активное сообщество D3.

Важно понимать, что для работы в Observable вы должны будете изучить способ работы с проектами D3, специфичный для этой платформы. Кроме того, вы не сможете напрямую копировать и вставлять блокноты Observable в среду разработки (хотя Observable 2.0 упрощает этот процесс). Поскольку в этой книге основное внимание уделяется построению визуализаций D3 в среде, близкой к используемой в промышленной разработке, мы не будем рассматривать блокноты Observable. Но для тех, кому это интересно, заметим, что существует отличная серия руководств, доступная по адресу https://observablehq.com/tutorials. Большинство методов и концепций, которые вы узнаете в этой книге, вполне можно перенести в блокноты Observable.

1.3 Передовые практики визуализации данных

Визуализация данных никогда не пользовалась такой популярностью, как в настоящее время. Богатством карт, диаграмм и сложных представлений систем и наборов данных отличаются не только рабочие места, но также наши развлечения и повседневная жизнь. С ростом популярности растут библиотеки визуализации данных и расширяются эстетические правила, способствующие удобочитаемости и пониманию. Наша аудитория, будь то широкая публика, научные работники или лица, принимающие решения, привыкла к представ-

лениям, когда-то считавшимся невероятно абстрактными и сложными. Это делает такие библиотеки, как D3, популярными не только среди специалистов по данным, но и среди журналистов, художников, ученых, ИТ-специалистов и даже энтузиастов визуализации данных.

Такое изобилие вариантов может ошеломить, а относительная простота изменения набора данных для отображения в виде потокового графика, дерева или гистограммы, как правило, подкрепляет мнение, что визуализация информации – это скорее стиль, чем содержание. К счастью, есть устоявшиеся правила, которые диктуют, какие диаграммы и методы использовать для разных типов данных. Эта книга не ставит своей целью охватить все передовые практики визуализации данных, но мы коснемся некоторых. Разработчики используют D3 для революционного использования цветов и макетов, но большинство стремится создавать визуальные представления данных, которые поддерживают решение практических задач.

При создании первых проектов визуализации часто лучше представить данные в виде гистограммы, чем в форме скрипичного графика, или в виде иерархической сетевой схемы (например, дендрограммы), чем в форме ориентированного графа. Визуально более сложные методы отображения, как правило, вызывают больше волнения, но они также могут заставить аудиторию сосредоточиться на эстетике, а не на самих данных. Нет ничего плохого в создании крутых и потрясающих визуализаций, но никогда не забывайте, что главная цель любой визуализации – поведать историю. Важным шагом является опрос людей с целью узнать, понимают ли они вашу визуализацию и как они ее интерпретируют. Нужны ли им объяснения? Какие выводы они могут сделать, взаимодействуя с вашим проектом? Рассказывает ли он историю, скрывающуюся за данными?

Чтобы правильно визуализировать информацию, вы должны знать, что можно, а чего нельзя делать. Вы должны иметь четкое представление о своих данных и своей аудитории. D3 дает огромную гибкость, но помните, что с большой силой приходит большая ответственность. Важно знать, какие диаграммы лучше подходят для представления определенного типа данных, но еще важнее помнить, что визуализации могут дезориентировать, если проектировать их без должной осторожности. Планируя начать разрабатывать собственные визуализации, изучите передовые методы визуализации данных.

Лучший способ узнать все это – познакомиться с работами признанных дизайнеров и практиков визуализации информации. Собрана целая библиотека этих работ, но мы приведем лишь несколько ресурсов, которые сочли наиболее полезными для начинающих. Это ни в коем случае не единственные источники информации для изучения визуализации данных, но они послужат отличной отправной точкой:

- *Better Data Visualizations* (издательство Колумбийского университета, 2021) Джонатана Швабиша (Jonathan Schwabish);

- *The Functional Art* (New Riders, 2013), *The Truthful Art* (New Riders, 2016) и *How Charts Lie*[1] (W. W. Norton, 2020) Альберта Кайро (Alberto Cairo);
- *Data Visualisation: A Handbook for Data Driven Design* (SAGE, 2019) Энди Кирка (Andy Kirk);
- *The Visual Display of Quantitative Information Envisioning Information* (Graphics Press, 2001) Эдварда Тафти (Edward Tufte);
- Design for Information (Rockport, 2013) Изабель Мейреллиш (Isabel Meirelles);
- *Pattern Recognition* (тезисы к диссертации, 2008, Школа дизайна Род-Айленда) Кристиана Суайнхарта (Christian Swinehart);
- *Visualization Analysis and Design*[2] (AK Peters, 2014) Тамары Манцнер (Tamara Munzner).

При знакомстве с источниками следует помнить, что литература часто фокусируется на статичных диаграммах. В проектах D3 вы будете создавать интерактивные и динамические визуализации. Поддержка взаимодействий может сделать визуализацию не только более читабельной, но и значительно более интересной. Пользователи, чувствующие, что занимаются исследованиями, а не чтением, даже если при этом просто щелкают мышкой, могут счесть содержимое визуализаций более убедительным и запоминающимся, чем если бы они просматривали статический эквивалент. Но эта дополнительная сложность требует изучения пользовательского опыта, о чем подробнее будет рассказываться в главе 7.

На этом мы завершаем первую главу! Мы еще не использовали D3, но вы уже получили все необходимые знания для начала работы. Возвращайтесь к этой главе, когда почувствуете неуверенность в выборе элемента SVG для использования в визуализациях, или если понадобится вспомнить, как манипулировать данными с помощью JavaScript. В следующей главе мы засучим рукава и создадим нашу первую визуализацию с помощью D3.

Интервью с Хешамом Эйссой и Линдси Поултер

Эйсса и Поултер – дизайнеры и разработчики визуализаций данных.

В одной из статей в вашем блоге вы написали, что до проекта Ben & Jerry's у вас обоих был опыт проектирования визуализаций данных в Tableau. Что заставило вас обратить внимание на D3? (https://heshameissa.com/blog/learn-d3/.)

---

[1] *Альберто Кайро*. Графики лгут. Как стать информационно грамотным человеком в мире данных? М.: БОМБОРА, 2019. ISBN: 978-5-04-169660-3. – *Прим. перев.*

[2] *Манцнер Тамара*. Визуализация данных. Полный и исчерпывающий курс для начинающих. М.: Эксмо, 2023. ISBN: ISBN 978-5-04-106797-7. – *Прим. перев.*

Мы с большим интересом изучили работу Надии Бремер (Nadieh Bremer) *The Pudding*, материалы, отмеченные наградой Information is Beautiful Awards, статьи различных новостных агентств по журналистике данных и отметили, что все понравившиеся нам проекты объединяло использование D3.

Кроме того, мы оба страстно желали использовать новые творческие способы отображения данных и не хотели ограничивать себя возможностями, которые поддерживались конкретным программным обеспечением. Познакомившись с D3 и технологиями веб-визуализации в целом, мы почувствовали, что с их помощью мы сможем создавать все, что только пожелаем. Единственным препятствием были лишь наши собственные способности!

В той же статье мне понравилось, как вы структурировали свой процесс обучения, учитывая, что у вас не было никакого опыта в разработке пользовательских интерфейсов, но было достаточно терпения идти вперед шаг за шагом. Что показалось вам самым сложным на этом пути обучения? Что понравилось больше всего?

Сложнее всего было изучить и понять огромное многообразие возможностей. Вдобавок ко всему мир разработки интерфейсов меняется так быстро, что нам также пришлось учиться не отставать и адекватно воспринимать примеры разных версий. Иногда мы обнаруживали, что слишком забегаем вперед, и старались набраться терпения, чтобы полностью изучить концепцию и только потом двигаться дальше. Это тоже было по-своему сложно.

Самый приятный момент наступил, когда мы наконец достигли умения напрямую применять наши знания к визуализации данных. Увидеть свою первую визуализацию SVG на странице было очень радостно для нас обоих. По мере исследования различных модулей D3 мы все отчетливей понимали, какие широчайшие возможности они нам дают. Модули, такие как d3-force, обладают большой гибкостью, и знакомство с ними действительно расширило наши представления о подходах к визуализации данных.

Где вы искали помощь, когда застревали?

Наличие партнера, который может задавать вопросы, лучше всего способствует обучению. Если мы чего-то не понимали, то задавали вопросы друг другу и в результате либо выясняли, что оба чего-то недопонимаем и нам нужно потратить на это больше времени, либо давали возможность одному из нас объяснить это.

Кроме того, мы тратили очень много времени на изучение проектов других специалистов в этой области и их способы реализации конкретных концепций. Одна из лучших черт сообщества специалистов по визуализации данных – готовность делиться знаниями. Примеры Майка Бостока (Mike Bostock) на Observable и вообще все сообщество Observable в целом не раз помогали нам. Обычно первым делом, занимаясь

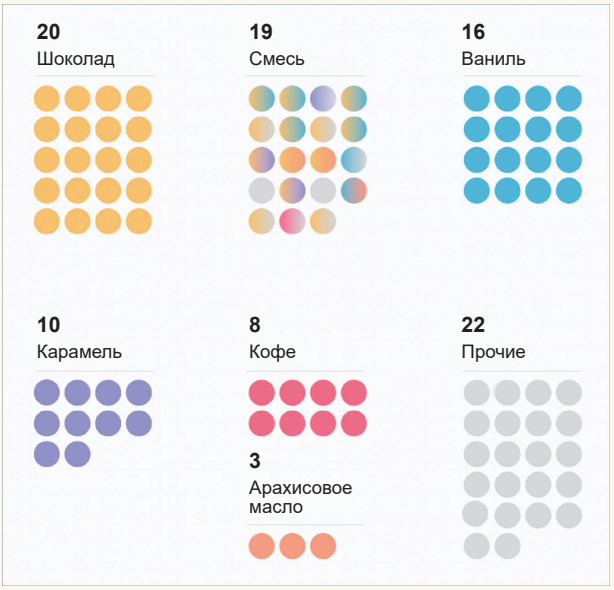
созданием новой диаграммы, мы обращались к его примерам и в 99 % случаев находили то, что нам было нужно. В книге *The Pudding* Надии Бремер (Nadieh Bremer) мы нашли множество примеров кода, поэтому мы часто спрашивали: «А как это реализовано у нее?» – и находили реальные рабочие примеры. Довольно забавно, что мы купили первое издание этой книги в один из выходных, когда бились над одной из концепций и не могли найти ответы в интернете.

Однако признаемся честно, что у нас были моменты, когда мы застревали по-настоящему. В большинстве случаев это было связано с ограниченностью нашего словарного запаса, и мы не могли четко и ясно выразить искомое, хотя в некоторых ситуациях мы действительно пытались найти ответы на очень специфические вопросы. Мы не боялись провести несколько дней (или недель) подряд в экспериментах, пробуя все подряд. В конечном счете тернии на пути изучения приемов визуализации данных заставили нас учиться и понимать (а не просто копировать и вставлять).

Есть ли что-то, что вам особенно нравится и/или не нравится в D3.js, по сравнению с другими инструментами визуализации данных?

D3 требует больше времени на изучение. Кроме того, иногда для создания столбчатой диаграммы может быть недостаточно несколько раз щелкнуть мышкой и нужно написать 50 и более строк кода. Однако это именно то, что нам нравится в D3, – мы можем контролировать все!

Представленный проект: «The Inside Scoop of Ben & Jerry's» (https://benjerry.heshlindsdataiz.com/)



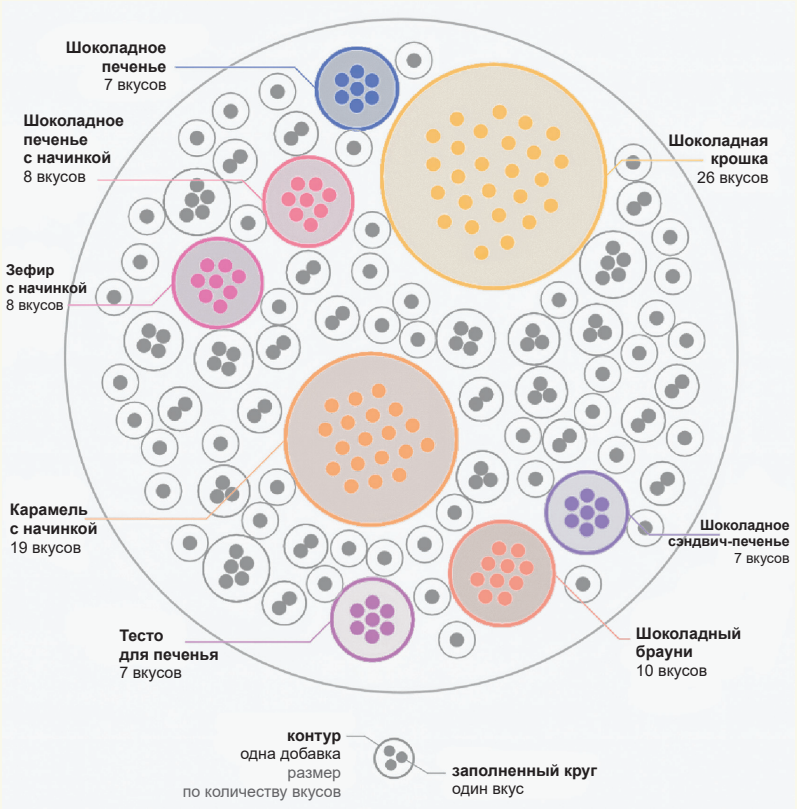
Проект «The 98 flavors of Ben & Jerry's»

Вы вместе работали над проектом для Ben & Jerry's. Как вы разделили работу? Один из вас больше сосредоточился на дизайне, а другой на разработке кода, или вы оба одинаково уделяли внимание этим двум аспектам?

Мы подошли к этому проекту не совсем традиционно, и, учитывая, что в тот момент как раз был пик самоизоляции во время пандемии короновируса, у нас было много времени, которое нужно было убить. Мы решили не разделять работу, а учиться вместе, и вместе сосредоточились на всем процессе от начала до конца. Мы вместе составляли сценарий и изучали данные, а затем ставили перед собой цели. Мы одновременно говорили: «Давай построим хордовую диаграмму, показывающую комбинацию вкусовых добавок для мороженого», – а потом по отдельности пытались выяснить, как ее построить. Закончив, мы сравнивали наш код, чтобы выяснить, кто из нас нашел наиболее удачный метод, объединяли наши находки и приступали к следующей задаче. Обретя чувство уверенности в своих силах, где-то примерно на полпути, мы начинали разделять и властвовать. Мы не всегда делили работу так, что один делал XYZ, а другой – ABC, мы оба просто делали то, что нужно было сделать!

В этом проекте используется графика, включающая и элементы SVG, и графику Canvas. Некоторые визуализации основаны на SVG, например «The Complete Ben & Jerry's Flavor Guide», другие в основном реализованы на Canvas, например «The 98 Flavors of Ben & Jerry's». А хордовая диаграмма в «The 79 Additions of Ben & Jerry's» вообще использует комбинацию обоих подходов. Как вы выбирали, когда использовать SVG, а когда Canvas? Вы рассматривали эту структуру с самого начала или выбрали Canvas во время реализации, возможно, ради повышения производительности?

Первоначально мы не рассматривали эту структуру! Честно говоря, мы даже не знали, что такое Canvas. Но, определившись с набором визуализаций, которые нужно создать, мы начали работать над реализацией и быстро поняли, что нам не хватает производительности. Это привело нас к исследованию других, более производительных вариантов, в ходе которого мы наткнулись на Canvas.



Хордовая диаграмма в проекте «The 79 Additions of Ben & Jerry's»

Вам удалось создать отличный проект для мобильных устройств, включающий богатые радиальные визуализации. Приходилось ли вам принимать сложные решения, например менять первоначальные идеи, чтобы адаптировать визуализации к небольшим сенсорным экранам? Есть ли у вас свои предпочтения между подходами «в первую очередь для мобильных устройств» и «в первую очередь для ПК», когда дело касается веб-визуализации?

Когда подошли к этапу «разделяй и властвуй», мобильные устройства были одними из последних, которые мы приняли во внимание, и на собственном горьком опыте усвоили, что откладывать их напоследок – не лучшее решение, потому что это потребовало идти на компромиссы. Нам приходилось решать не только что изменить, но и как это сделать. Использование scrollytelling (http://mng.bz/g7MV) помогло нам с созданием реализации для мобильных устройств, потому что это дало нам возможность прокручивать текст поверх визуальных элементов и заменить взаимодействия выделением важных точек данных.

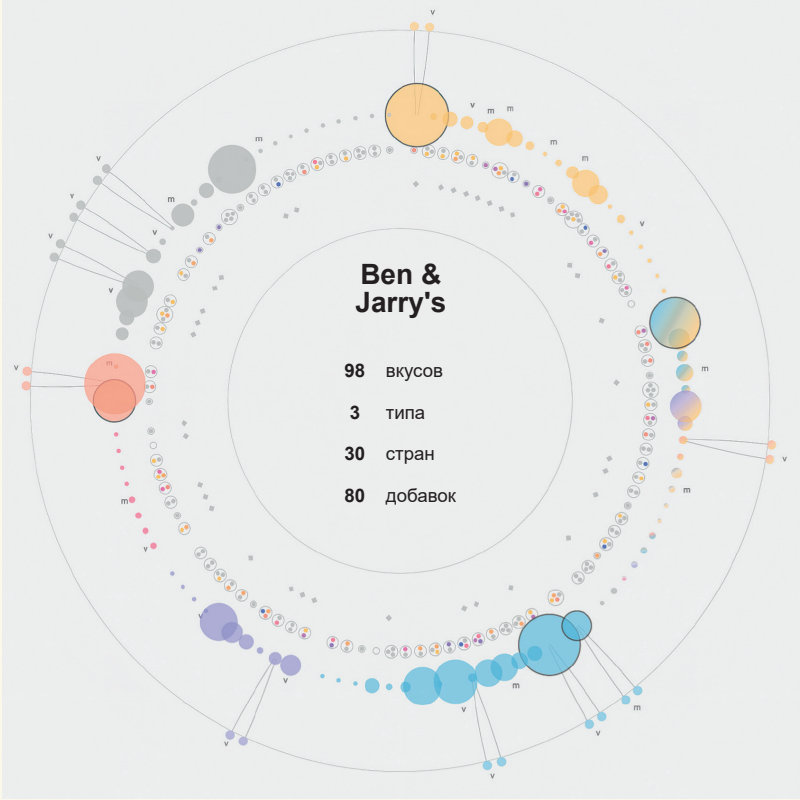
Подход «в первую очередь для ПК» дает гораздо больше пространства для игры, позволяет создавать более сложные взаимодействия и проек-

ты. Однако необходимость поддержки мобильных устройств заставляет сводить каждую визуализацию к ее основной цели, что может быть полезно для достижения максимальной ясности.

Сколько примерно времени заняло создание этого проекта? Были ли этапы значительно длиннее или сложнее других?

Учитывая, что это был наш первый настоящий проект D3, параллельно с которым мы продолжали учиться, и мы переделывали каждый раздел по нескольку раз, чтобы приблизиться к идеалу, нам потребовалось около четырех полных месяцев работы без выходных. В конце концов нам пришлось установить крайний срок, потому что мы знали, что если мы этого не сделаем, то продолжим переделывать, дорабатывать и находить новые проблемы.

Самыми сложными и продолжительными были этапы, которые больше касались разработки пользовательского интерфейса и меньше самой визуализации. У нас был опыт разработки визуализаций, и мы подходили к их созданию со знанием дела, но мы поняли, что нам придется потратить время, чтобы заставить все это работать на всех устройствах и браузерах.



Проект «The Complete Ben & Jerry's Flavor Guide»

Есть ли какая-то конкретная функция или модуль D3, которые вы хотели бы изучить и/или освоить лучше в ближайшем будущем?

Конечно! Кое-что мы используем по принципу «скопировал и вставил», не понимая до конца, как это работает. И модули `chords`, `sankeys` и `geo` в их числе! Кроме того, освоение большего количества математических функций в целом может помочь нам лучше использовать возможности D3.

Итоги

- D3 – это инструмент, который пригодится всем, кому нужна полная творческая и техническая свобода при создании визуализаций данных.
- Приложения D3 оформляются и обслуживаются подобно традиционным веб-страницам.
- D3 никогда не используется отдельно, а является частью экосистемы технологий и инструментов, которые используем для создания богатых веб-интерфейсов: HTML, CSS, JavaScript, SVG, Canvas и фреймворков, таких как React или Svelte.
- При создании визуализаций данных чаще всего используются такие фигуры SVG, как линии, прямоугольники, круги, эллипсы, кривые и текст.
- Для работы с D3 вам потребуется базовое понимание этих форм и их основных атрибутов.
- Прежде чем начинать использовать D3, следует освоить два приема программирования на JavaScript – объединение вызовов методов в цепочки и манипулирование объектами:
  - цепочка вызовов методов – это шаблон программирования, когда несколько методов одного и того же объекта вызываются друг за другом;
  - в D3 наборы данных часто структурируются как массивы объектов. JavaScript предлагает несколько методов для доступа к данным внутри этих структур.
- Для разработчиков на D3 важно постоянно изучать передовые практики визуализации данных. Существует множество ресурсов, которые могут помочь вам начать обучение.

2 Манипулирование деревом DOM

https://t.me/it_boooks/2

В этой главе рассматриваются:

- настройка локальной среды разработки проектов D3;
- выборка элементов из дерева DOM;
- добавление элементов HTML или SVG в выбранный фрагмент;
- добавление и изменение атрибутов и стилей элементов DOM.

Теперь, обсудив экосистему D3, можно приступать к работе! В этой главе мы заложим основы нашей первой визуализации и одновременно узнаем, как манипулировать деревом объектной модели документа (DOM) с помощью D3.

Манипулирование деревом DOM – основополагающая функция D3, и методы, с которыми вы здесь познакомитесь, станут для вас, как для разработчика на D3, самыми часто используемыми (если только вы не решите использовать фреймворк JavaScript, такой как React или Svelte, которые мы обсудим в главе 8). Сначала рассмотрим операцию выбора, позволяющую извлекать один или несколько элементов из DOM. Вы увидите, что эта операция в D3 выглядит очень просто и понятно. После получения выборки с ней обычно выполня-

ются какие-то другие операции. В проектах D3 в выборки регулярно добавляются новые элементы HTML или SVG. Например, создавая визуализации, мы часто добавляем фигуры в контейнеры SVG. Наконец, после добавления новых элементов мы настраиваем их позиции, размеры и цвета, устанавливая их атрибуты и стили.

Эта книга посвящена созданию проектов в локальной среде разработки, поэтому давайте создадим такую среду, прежде чем мы погрузимся в знакомство с методами D3. В разделе 2.2 мы покажем, как использовать Visual Studio Code (VS Code) и его расширение Live Server, чтобы за несколько минут настроить локальную среду, готовую к работе.

2.1 Наша первая визуализация D3

В этой и следующей главах мы разработаем нашу первую визуализацию D3: столбчатую диаграмму, показанную на рис. 2.1. Мы упоминали в главе 1, что D3, пожалуй, не самый эффективный инструмент для создания простых классических диаграмм, тем не менее столбчатая диаграмма идеально подходит для знакомства с базовыми концепциями D3. Следуйте за нами, и вскоре вы заложите прочные основы, которые позволят вам с легкостью создавать сложные визуализации.

В основе нашей столбчатой диаграммы лежат данные, взятые из отчета о состоянии сферы визуализации данных в 2021 году, подготовленного сообществом специалистов по визуализации данных (www.datavisualizationsociety.org).

В опросе приняли участие 2181 специалист по визуализации данных – от профессионалов до студентов и любителей. Наша задача – визуализировать ответы на один вопрос: «Какие технологии вы чаще всего используете для визуализации данных?» Респонденты могли выбирать любые инструменты, перечисленные в заранее определенном списке. На рис. 2.1 показана диаграмма, в которой по вертикали перечислены инструменты, а горизонтальными столбцами отражено количество респондентов, выбравших этот инструмент. Согласно опросу, D3 замыкает десятку наиболее популярных инструментов визуализации данных. А теперь приступим!

> **ПРИМЕЧАНИЕ** Общество специалистов по визуализации данных ежегодно проводит опрос о состоянии отрасли. Самые свежие сведения о нашей отрасли вы найдете на сайте www.datavisualizationsociety.org/survey.

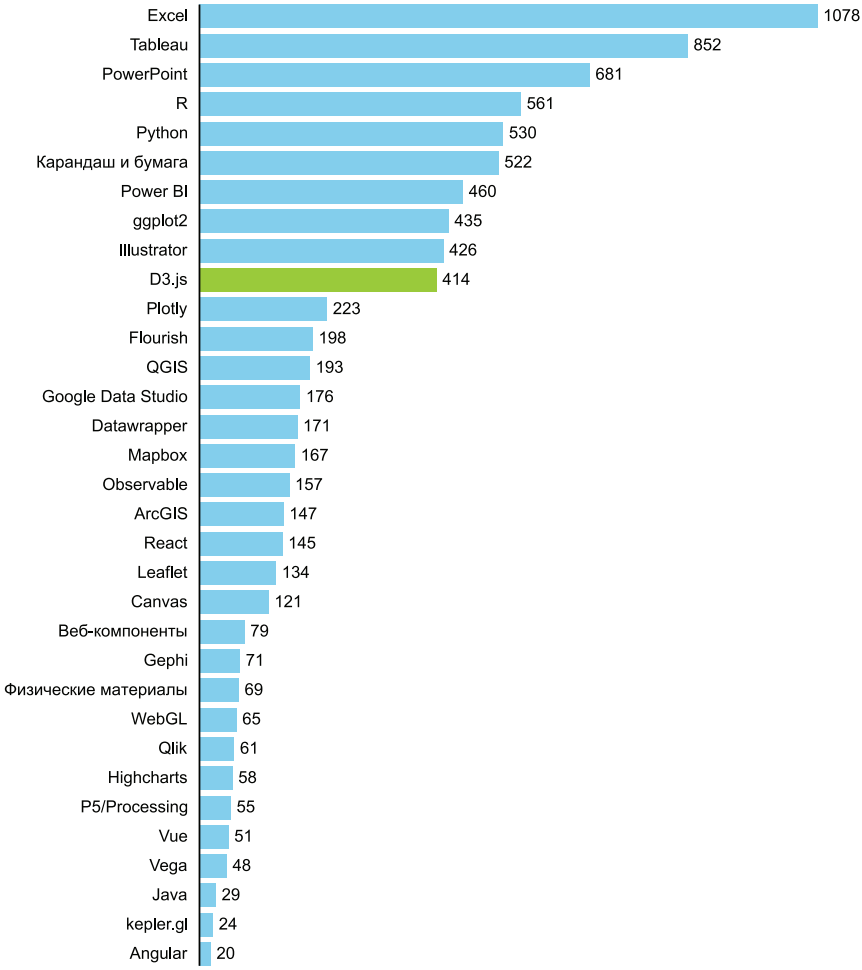


Рис. 2.1 Наиболее популярные технологии визуализации данных. В этой главе мы начнем строить эту столбчатую диаграмму (источник: отчет о состоянии сферы визуализации данных в 2021 году, подготовленный сообществом специалистов по визуализации данных)

2.2 Подготовка среды

Прежде чем начать использовать D3, нужно решить, где мы будем создавать и запускать наши проекты. Мы можем воспользоваться онлайн-редактором кода, например Observable (https://observablehq.com) или CodePen (https://codepen.io), которые отлично подходят для быстрого тестирования и распространения кода. Но поскольку цель

этой книги – помочь вам подготовиться к созданию проектов D3 для веб-сайтов и веб-приложений, мы выберем локальную среду разработки.

Если идея заняться настройкой среды разработки заставляет поежиться, то не волнуйтесь. Уже давно канули в Лету времена, когда приходилось тратить на этот процесс по полдня. Благодаря современным инструментам все необходимые настройки можно выполнить в первый раз за несколько минут и потом приступить к работе одним нажатием кнопки.

Возможно, у вас уже возник вопрос: «Почему нельзя просто открывать HTML-файлы в браузере, как мы делали это, создавая галерею фигур SVG в главе 1?» Иногда этот подход работает просто отлично, но в конечном итоге он приводит к тому, что браузер отказывается выполнять определенные задачи и выдает ошибки. Некоторые браузеры запрещают загрузку локальных файлов с JavaScript по соображениям безопасности и требуют загружать их через веб-сервер. В проектах D3 почти всегда приходится загружать файл данных, поэтому нам нужен веб-сервер.

На протяжении всей книги мы будем использовать редактор VS Code, часто в сочетании с расширением Live Server, которое предоставляет локальный веб-сервер. Но если у вас уже имеется настроенная среда, смело используйте ее и переходите к разделу 2.2.1.

> **ПРИМЕЧАНИЕ** Структура проекта, которую мы будем использовать в первой части этой книги, тривиальна. Наша цель – сделать среду максимально простой, чтобы сосредоточиться на изучении D3. Но если вы имеете опыт разработки и хотите работать с модульными проектами, вы вполне можете это сделать. Соответствующие инструкции по установке D3 с помощью Node Package Manager (NPM) и импорту библиотек в файлы вы найдете в разделе 8.2.

В среде разработчиков большой популярностью пользуется VS Code – бесплатный редактор с открытым исходным кодом, простой в использовании и при этом очень мощный. Он имеет встроенную поддержку Git (избавляя от необходимости открывать окно терминала) и обладает широкими возможностями настройки. Если вы еще не установили VS Code, то сделайте это прямо сейчас, загрузив дистрибутив с веб-сайта VS Code по адресу https://code.visualstudio.com/Download. После установки VS Code вам нужно будет установить его расширение Live Server. Если вам нужна помощь в этом, то обратитесь к приложению А, где также объясняется, как с помощью расширения Live Server запускать и останавливать локальный веб-сервер.

ПРИМЕЧАНИЕ Не забудьте загрузить файлы с примерами кода для этой книги из репозитория на GitHub (http://mng.bz/Xqjv). В этой главе мы будем работать с папкой `start`. Если вы растеряетесь и захотите посмотреть решение, то вы найдете его в папке `end`. Работая с главой, открывайте в редакторе только одну папку, `start` или `end`. Если открыть сразу обе и использовать расширение Live Server для обслуживания проекта, то некоторые пути не будут работать, особенно это касается загрузки набора данных в проект. Файлы с кодом распределены по разделам, поэтому, переходя к новому разделу в главе, вы можете продолжить работу с уже открытыми файлами кода или начать заново с папки, предназначенной для данного раздела. Оба варианта приведут к одному и тому же результату.

2.2.1 Структура нашего первого проекта D3

Проекты D3, над которыми мы будем работать в следующих главах, будут иметь структуру, похожую на представленную на рис. 2.2.

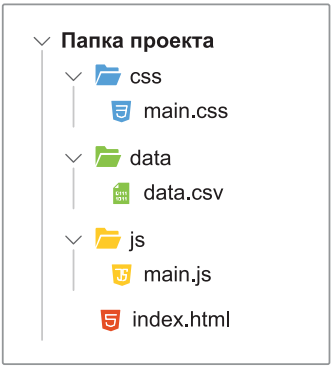


Рис. 2.2 Структура папок нашего первого проекта D3

В корне проекта находится файл `index.html`, содержащий начальную разметку нашего проекта. Здесь же мы загрузим библиотеку D3, файлы JavaScript и CSS. Помимо этого файла у нас есть три папки:

- папка `/css` содержит все файлы CSS, имеющие отношение к проекту. Эта книга не о CSS, тем не менее мы будем использовать каскадные таблицы стилей время от времени. Для простоты мы также сгруппируем наши стили в минимальное количество файлов. Но имейте в виду, что в профессиональных проектах папка CSS может иметь гораздо более сложную организацию, и часто используются препроцессоры CSS, такие как Syntactically Awesome Stylesheets (SASS) и Leaner Stylesheets (LESS);
- папка `/data` содержит наборы данных. В проекте столбчатой диаграммы набор данных представлен файлом CSV. Каждая

строка в этом файле содержит название технологии, за которой следует число, отражающее, сколько раз респонденты выбрали эту технологию;
- наконец, папка `/js` содержит файлы JavaScript. Для простоты мы поместим весь наш код в один файл `main.js`. Но позже мы поговорим о том, как можно разделить код на несколько файлов или компонентов для улучшения поддержки и тестирования.

2.2.2 Загрузка D3 в проект

Перед началом проверьте, работает ли ваш веб-сервер, посмотрев в правый нижний угол окна VS Code. Вы должны увидеть надпись «Port 5500» (или другой номер порта), указывающую, что сервер работает. Если вместо этого вы видите надпись «Go Live», то щелкните на ней, чтобы запустить сервер. Теперь ваш проект можно открыть в браузере по адресу `http://localhost:5500/` (или с другим номером порта, который использует сервер).

После запуска веб-сервера нужно сделать еще кое-что, прежде чем начать писать код: загрузить библиотеку D3 в проект. В этой книге мы будем использовать два основных подхода. Первый – добавление в `index.html` тега `<script>`, который ссылается на последнюю версию D3. Этот подход можно использовать для загрузки всей библиотеки D3 или только определенных модулей. Второй – загрузка D3 в виде модулей NPM. Этот подход лучше подходит для сайтов, созданных с помощью React или других фреймворков JavaScript.

В этой главе мы используем первый подход как самый простой. Но по мере продвижения вперед начнем использовать второй как более часто используемый в современных профессиональных проектах D3.

В VS Code или в другом редакторе кода, выбранном вами, откройте файл `index.html`, расположенный в корне папки. Прямо перед закрывающим тегом `</body>` загрузите версию 7 библиотеки D3 (последнюю на момент написания этой книги), используя тег `script`. Добавьте еще один тег `script` для загрузки файла `main.js`, расположенного в папке `/js`, и сохраните проект. В листинге 2.1 показано, что у вас должно получиться:

```
<script src="https://d3js.org/d3.v7.min.js"></script>
<script src="js/main.js"></script>
```

Браузер загружает файлы JavaScript в том порядке, в каком теги `script` перечислены в `index.html`. Мы должны загрузить библиотеку D3 раньше, чем `main.js`, иначе браузер не будет иметь доступа к методам D3, используемым в `main.js`. Он сообщит об ошибках и не выполнит код.

Кроме того, сценарии должны загружаться в последнюю очередь, поэтому теги `script` следует поместить непосредственно перед закрывающим тегом `</body>`. При таком подходе мы сокращаем время загрузки нашей страницы, так как она не должна ждать, пока выполнятся сценарии, а также гарантируем доступность дерева DOM к моменту, когда сценарии попытаются манипулировать им.

> **ПРИМЕЧАНИЕ** Для локальной разработки можно использовать неминифицированную версию библиотеки (https://d3js.org/d3.v7.js), чтобы получить доступ к читаемой версии исходного кода. Но по окончании разработки и перед отправкой кода в производство мы рекомендуем заменить ее на минифицированную версию, чтобы повысить производительность.

Листинг 2.1 Загрузка библиотеки D3 в тег script: index.html

```
<!DOCTYPE html>
<html>
<head> ... </head>
<body>
  <div class="container">
    <h1>You are about to start working with D3!</h1>
  </div>

  <script src="https://d3js.org/d3.v7.min.js"></script>

  <script src="js/main.js"></script>

</body>
</html>
```

Тег script, загружающий минифицированную библиотеку D3

Тег script, загружающий файл main.js

Закрывающий тег </body>

Теперь проверим, правильно ли загрузились библиотека D3 и файл `main.js`. В редакторе кода перейдите в папку `/js` и откройте `main.js`. Вставьте в `main.js` следующий фрагмент и сохраните файл:

```
d3.select("h1").style("color", "plum");
```

В следующем разделе мы подробно объясним, что делают вызовы методов D3 в этом фрагменте кода, но если говорить кратко, то они выбрали заголовок `h1` и изменили его цвет на цвет с именем `"plum"`. Если теперь открыть проект в браузере, то можно заметить, что цвет заголовка изменился (рис. 2.3).



Рис. 2.3 Цвет заголовка был изменен с помощью методов D3

Теперь, убедившись, что D3 загрузилась в проект, можно удалить только что вставленный фрагмент из `main.js` и заголовок `h1` из `index.html`. В следующем разделе мы покажем, как с помощью D3 выбирать элементы в дереве DOM.

> **ПРИМЕЧАНИЕ** Выбор элементов дерева DOM и управление ими реализованы в D3 несколько старомодно. В настоящее время проекты веб-интерфейсов обычно реализуются с помощью фреймворков, таких как React или Svelte, и на них перекладывается обработка изменений в DOM. Мы демонстрируем эти старомодные методы, потому что их знание поможет вам лучше понять, как работает D3, и потому что, скорее всего, вы будете использовать их в небольших проектах. Но начиная с главы 8 мы будем использовать более современный подход.

2.3 Выбор элементов

В проектах D3 мы постоянно манипулируем деревом DOM, и любая манипуляция DOM начинается с выбора. Выбор похож на захват элемента из DOM и приведение его в готовность к дальнейшим манипуляциям. D3 имеет два метода для выбора элементов: `d3.select()` и `d3.selectAll()`.

Метод `d3.select()` принимает параметр с селектором и возвращает первый найденный элемент, соответствующий этому селектору. Этот метод связан с объектом `d3` и используется для выбора одного элемента. Как показано на рис. 2.4, селектор может быть атрибутом класса, идентификатором, именем тега или любой их комбинацией – точно так же, как селекторы CSS.

```
d3.select("селектор");
```

(строка)

идентификатор	`"#my-id"`
класс	`".my-class"`
имя тега	`"h1"`
	`"svg"`
комбинация селекторов	`"circle.faded"`
	`"#my-id p"`
	`"rect, circle"`

Рис. 2.4 Метод `select()`

Давайте возьмем для примера вымышленное дерево DOM, изображенное на рис. 2.5. Оно содержит элемент `div` с заголовком `h1`, элемент абзаца `p` с классом `intro` и еще один элемент `div` с идентификатором `viz-container`. Этот элемент `div` обертывает другой абзац и контейнер SVG. Наконец, контейнер SVG включает три элемента `circle`. Первый и последний из этих элементов `circle` имеют класс `faded`.

```
<div>
  <h1>A title</h1>
  <p class="intro"> ... </p>
  <div id="viz-container">
    <p> ... </p>
    <svg>
      <circle class="faded" />
      <circle />
      <circle class="faded" />
    </svg>
  </div>
</div>
```

Рис. 2.5 Пример вымышленного дерева DOM

Чтобы выбрать заголовок `h1`, мы можем передать в вызов `d3.select()` имя тега:

```
d3.select("h1");
```

Аналогично выбрать абзац с классом `intro` или элемент `div` с идентификатором `viz-container` можно, используя соответствующий класс или идентификаторы. Так же как в селекторах CSS, имена классов предваряются точкой (`.`), а идентификаторы – хештегом (`#`):

```
d3.select(".intro");
d3.select("#viz-container");
```

Также можно использовать комбинацию селекторов. Например, чтобы выбрать элемент абзаца внутри `div` с идентификатором `viz-container`, можно объединить два селектора через пробел:

```
d3.select("#viz-container p");
```

Такой комбинированный селектор даст тот же эффект, что и объединение двух методов `select()`:

```
d3.select("#viz-container").select("p");
```

Важно помнить, что метод `d3.select()` возвращает только первый встретившийся элемент DOM, соответствующий указанному селектору. Например, в нашем примере, показанном на рис. 2.6, есть три элемента `circle`. Но вызов `d3.select("circle")` вернет только первый элемент и игнорирует остальные.

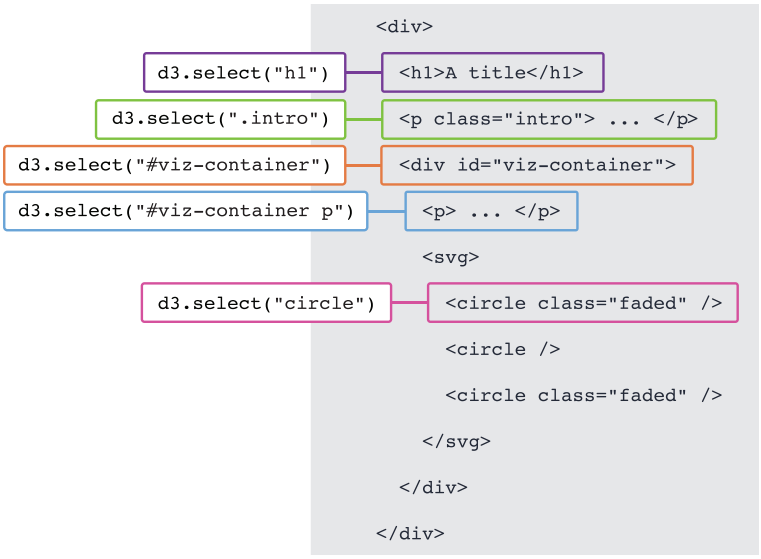


Рис. 2.6 Элементы DOM, возвращаемые методом `d3.select()`

А можно ли получить сразу несколько элементов? Да, можно. Для этой цели существует метод `d3.selectAll()`. Он действует подобно методу `d3.select()`, но возвращает все элементы DOM, соответствующие заданному селектору. Например, если вернуться к нашему вымышленному примеру DOM, то вызов `d3.selectAll("circle")` вернет все элементы `circle`, содержащиеся в DOM, как показано на рис. 2.7.

Иногда также полезно помнить, что, как и в CSS, можно сгруппировать несколько селекторов, разделив их запятыми. Например, следующий вызов выберет заголовок `h1` и абзац с классом `intro`, как показано на рис. 2.8:

```
d3.selectAll("h1, .intro");
```



Рис. 2.7 Элементы DOM, возвращаемые методом `d3.selectAll()`

В большинстве случаев бывает желательно сохранить выбранные элементы в константах JavaScript, чтобы повторно использовать их позже. Выбранные элементы можно сохранять так же, как любые другие константы JavaScript (`const`) или переменные (`let`):

```
const myCircles = d3.selectAll("circle");
```



Рис. 2.8 Группировка селекторов через запятую

Доступ к документации по модулям D3

D3 состоит из набора модулей, которые можно использовать по отдельности и в разных комбинациях в зависимости от потребностей. Каждый модуль содержит несколько методов, решающих связанные задачи.

Все методы, обсуждаемые в этой главе, являются частью модуля `d3-selection`. Этот модуль описан на официальном сайте документации D3 (https://d3js .org/), надежном и всегда актуальном ресурсе.

Если вы только осваиваете веб-разработку, то такая документация может поначалу напугать, но постепенно вы начнете лучше понимать ее технический язык.

2.4 Добавление элементов в выборку

Выборки хороши, но они не приносят особой пользы, если с ними ничего не делать. Типичный шаблон в D3 – выбрать элементы, чтобы добавить в них другие элементы. Чистый JavaScript позволяет добавлять элементы, но D3 значительно упрощает эту операцию.

Основным методом D3, используемым для добавления элементов, является `selection.append()`. Он принимает тип элемента или имя тега и добавляет его *в конец* выбранного элемента (рис. 2.9).

```
selection.append("тип");
```

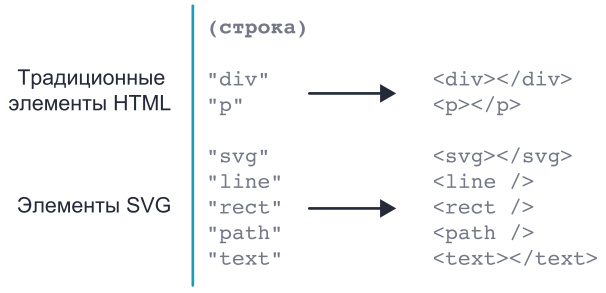


Рис. 2.9 Метод `append()`

Вернемся к нашему примеру DOM. Чтобы добавить элемент прямоугольника как последний дочерний элемент в контейнер SVG, мы сначала должны выбрать контейнер SVG, а затем вызвать метод `append()` полученной выборки (рис. 2.10). Тип добавляемого узла – `rect` – передается методу `append()` в виде параметра:

```
d3.select("svg").append("rect");
```

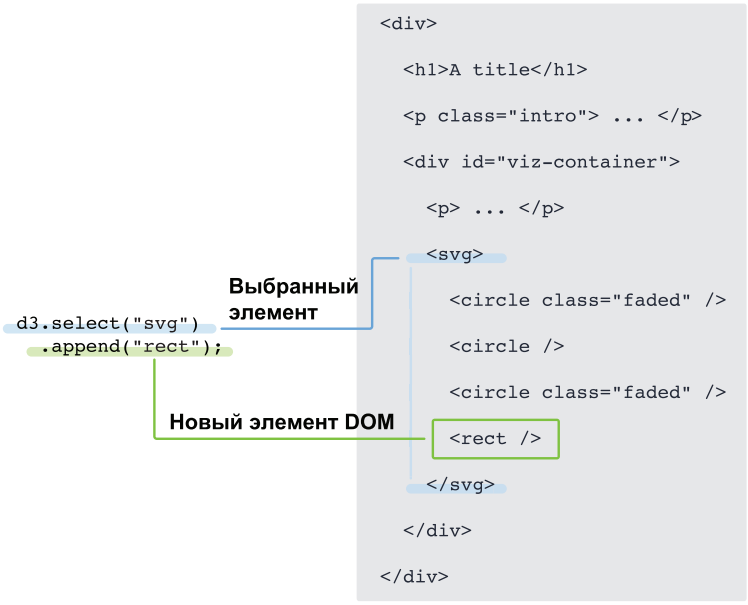


Рис. 2.10 Вызов `selection.append()` для добавления элемента в выборку как последнего потомка

Также можно было бы использовать вызов `d3.selectAll("div")`, чтобы выбрать все узлы `div` и в каждый добавить элемент абзаца, как показано на рис. 2.11:

```
d3.selectAll("div").append("p");
```

Чтобы применить полученные знания на практике, приступим к построению столбчатой диаграммы, описанной в начале этой главы.

Откройте в редакторе папку `start` в каталоге с файлами для главы 2 и запустите локальный веб-сервер. Если вы не помните, как запустить веб-сервер с расширением Live Server в VS Code, то обратитесь к приложению А. Откройте в редакторе файл `index.html`; обратите внимание, что он содержит элемент `div` с классом `responsive-svg-container`.

Как обсуждалось в главе 1, большинство визуализаций D3 основаны на использовании элементов SVG, и наша столбчатая диаграмма не является исключением. Поэтому нам понадобится контейнер SVG, в который мы поместим фигуры SVG. Добавим этот контейнер.

Откройте файл `main.js` в папке `/js`. Используя метод `d3.select()`, выберите `div` с классом `responsive-svg-container` и добавьте внутрь этого `div` элемент SVG. Следующий фрагмент кода показывает, как это сделать:

```
d3.select(".responsive-svg-container")
  .append("svg");
```

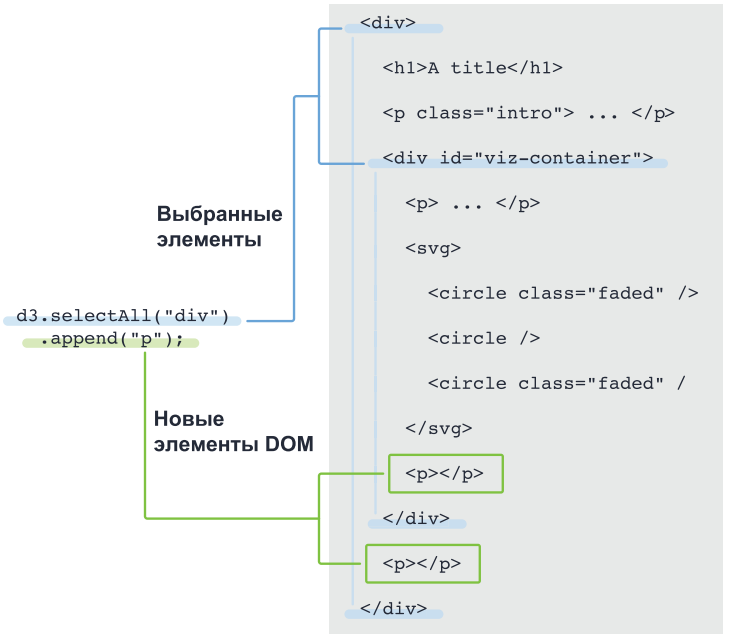


Рис. 2.11 В комбинации с `d3.selectAll()` метод `append()` добавит указанный элемент во все узлы, попавшие в выборку

Сохраните файл `main.js` и посмотрите на проект в браузере. В области просмотра не видно никаких изменений, но если открыть панель инспектора, как показано на рис. 2.12, то можно заметить, что элемент SVG был добавлен в DOM, как мы и хотели!

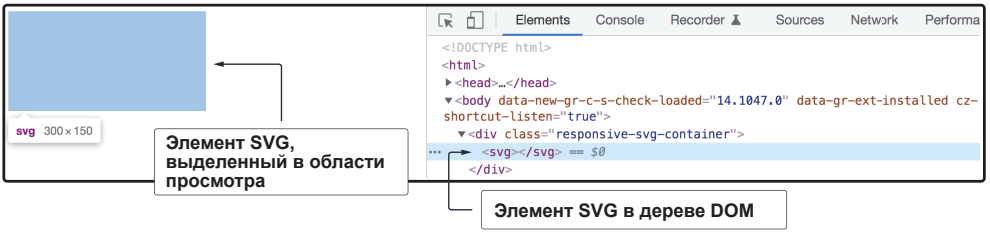


Рис. 2.12 Элемент SVG добавлен в дерево DOM

В следующем разделе мы сделаем элемент SVG адаптивным, присвоив ему атрибут `viewBox`.

2.5 Установка и изменение атрибутов

В главе 1 мы подробно обсудили основные элементы SVG и атрибуты, определяющие их местоположение и размер. Мы также объяс-

нили, что вам как разработчику на D3 придется устанавливать и изменять эти атрибуты в своем коде. Теперь самое время узнать, как это сделать!

Атрибуты можно устанавливать и изменять с помощью метода `selection.attr()`, где имя «attr» означает «attribute» (атрибут). Как показано на рис. 2.13, метод `attr()` принимает два параметра: имя атрибута и его значение. Значение также можно установить напрямую или через функцию доступа, как будет рассказываться в главе 3.

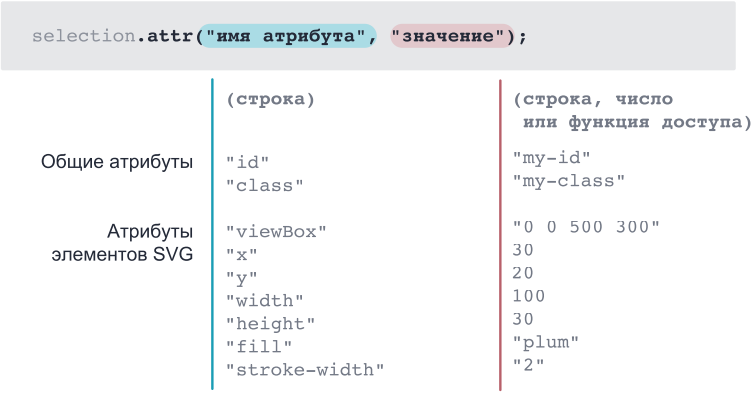


Рис. 2.13 Метод `attr()`

В нашем упражнении со столбчатой диаграммой элемент `div`, вмещающий контейнер SVG, имеет класс `responsive-svg-container`. Открыв файл `main.css` в папке `/css`, вы увидите, что этот класс применяет все стили, необходимые для создания адаптивного контейнера SVG, как обсуждалось в разделе 1.2.2. Здесь контейнер имеет свойство `max-width`, равное 1200 пикселей, которое также определяет максимальную ширину столбчатой диаграммы.

Чтобы контейнер SVG сохранял пропорции при адаптации к своему контейнеру, нужно задать его атрибут `viewBox`. Для этого вызовем метод `attr()`. Как показано ниже, в первом параметре методу `attr()` передается имя атрибута, в данном случае `viewBox`. Обратите внимание, что буква «B» в имени этого атрибута – заглавная. Атрибуты, определяющие параметры представления, чувствительны к регистру, поэтому важно соблюсти нотацию CamelCase в имени атрибута `viewBox`, чтобы браузер мог его распознать.

Во втором параметре передается значение атрибута `viewBox` – список из четырех чисел. Первые два числа – это начало системы координат `viewBox`, расположенное в точке (0,0); последние два числа – это ширина и высота `viewBox`. Ширина соответствует свойству `max-width` контейнера `div`, т. е. 1200 пикселей, а высоту зададим равной 1600 пикселей. Мы скорректируем ее позже, если понадо-

бится. Соответственно, мы присваиваем атрибуту `viewBox` значение `"0 0 1200 1600"`:

```
d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600");
```

Установив атрибут `viewBox` элемента SVG, сохраните `main.js` и взгляните на свой проект в инспекторе. Вы увидите, что в элементе SVG появился атрибут `viewBox` (рис. 2.14). Кроме того, если теперь попробовать уменьшить область просмотра браузера, элемент SVG адаптируется, сохраняя при этом соотношение сторон 1200:1600.

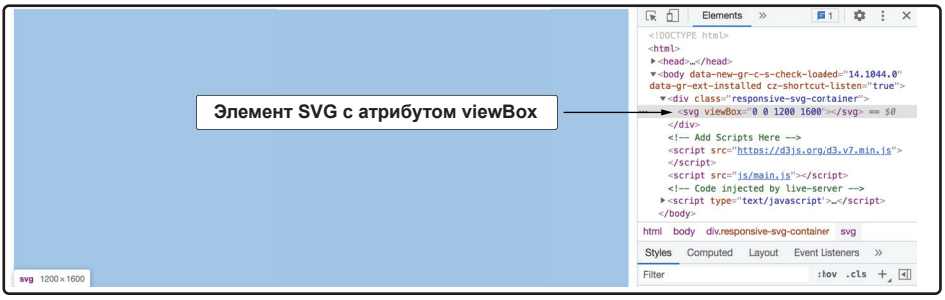


Рис. 2.14 Элемент SVG с атрибутом viewBox

Теперь давайте сохраним результат последнего вызова в константу с именем `svg`, которую вскоре мы используем повторно:

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600");
```

Каждый раз, добавляя новый элемент в выборку вызовом метода `.append()`, мы изменяем элемент, возвращаемый выборкой. Например, когда мы повторно используем константу `svg`, она вернет не элемент `div` с классом `responsive-svg-container`, а контейнер SVG, который мы только что добавили.

Соглашение об отступах

Прежде чем продолжить, обсудим соглашение об отступах, принятое в сообществе D3. В наших последних примерах кода вы могли заметить, что каждый следующий вызов метода в цепочке записывается на новой строке. Это улучшает читаемость, особенно когда в цепочке больше двух-трех вызовов методов. Вы также могли заметить, что перед вызовом метода `append()` добавлен отступ в два пробела, а перед `attr()` – четыре пробела. В соответствии с соглашением об отступах, принятым

в сообществе D3, каждый следующий уровень вложенности получает дополнительный отступ в два пробела.

В D3 каждый раз, когда мы добавляем новый элемент в выборку, мы обновляем элементы DOM, входящие в выборку. Добавляя атрибуты и стили во вновь добавленные элементы, правильный отступ помогает нам видеть, к какой выборке применяются эти атрибуты и стили.



Соглашение об отступах в D3

Удобство соглашения об отступах особенно отчетливо проявляется, когда производится добавление нескольких элементов подряд. Представьте, что после добавления элемента SVG и установки атрибута `viewBox` мы добавляем в контейнер SVG элемент группировки с классом `my-group`. Затем мы добавляем элемент прямоугольника в группу и добавляем в него необходимые атрибуты. Как можно видеть на предыдущем рисунке, для этого достаточно объединить несколько вызовов методов в цепочку. А благодаря соглашению об отступах эта цепочка легко читается, и мы сразу видим, к какой выборке или элементу применяется каждый атрибут.

Столбчатые диаграммы, подобные той, которую мы строим, состоят из прямоугольников SVG, которые создаются с помощью элемента `rect`. Просто чтобы попрактиковаться в выборе элементов и применении метода `attr()`, добавим один прямоугольник в нашу диаграмму, который будет представлять количество специалистов по визуализации, выбравших инструмент D3.js. Если вы откроете файл `data.csv` в папке `/data`, то увидите, что в ходе опроса 414 специалистов заявили, что регулярно используют D3.

Откроем файл `main.js` и начнем с обращения к константе `svg`, которая возвращает контейнер SVG. Добавим элемент `rect` в контейнер SVG. Сохраним проект и проверим (в панели инспектора), что элемент `rect` действительно добавился внутрь SVG:

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600");

svg
  .append("rect");
```

Элемент `rect` был добавлен в дерево DOM, но пока не виден на экране, потому что его обязательные атрибуты не установлены. Мы знаем, что для появления прямоугольника SVG на экране необходимо задать четыре его атрибута. Подробнее об этом рассказывается в разделе 1.2.2. Атрибуты `x` и `y` управляют местоположением верхнего левого угла прямоугольника. Давайте пока поместим его в точку `(10, 10)`. Ширина прямоугольника должна быть пропорциональна числу специалистов (414), отдавших предпочтение D3, а высота может быть любой – мы будем использовать 16. Присвоив атрибутам `width` и `height` значения 414 и 16 соответственно, мы получим прямоугольник с шириной 414 пикселей и высотой 16 пикселей. Значения этих четырех атрибутов передаются как числа:

```
svg
  .append("rect")
    .attr("x", 10)
    .attr("y", 10)
    .attr("width", 414)
    .attr("height", 16);
```

Наконец, присвоим атрибуту `fill` прямоугольника имя цвета «бирюзовый», и он передается в виде строки:

```
svg
  .append("rect")
    .attr("x", 10)
    .attr("y", 10)
    .attr("width", 414)
    .attr("height", 16)
    .attr("fill", "turquoise");
```

Обратите внимание, как здесь используется соглашение об отступах: для обозначения новый выборки, созданной при добавлении прямоугольника, используется отступ в два пробела, а для записи вызовов метода `attr()` – четыре пробела. Благодаря этому мы ясно видим, что атрибуты устанавливаются в элемент `rect`. После сохра-

нения проекта прямоугольник будет виден в области просмотра браузера, как показано на рис. 2.15.

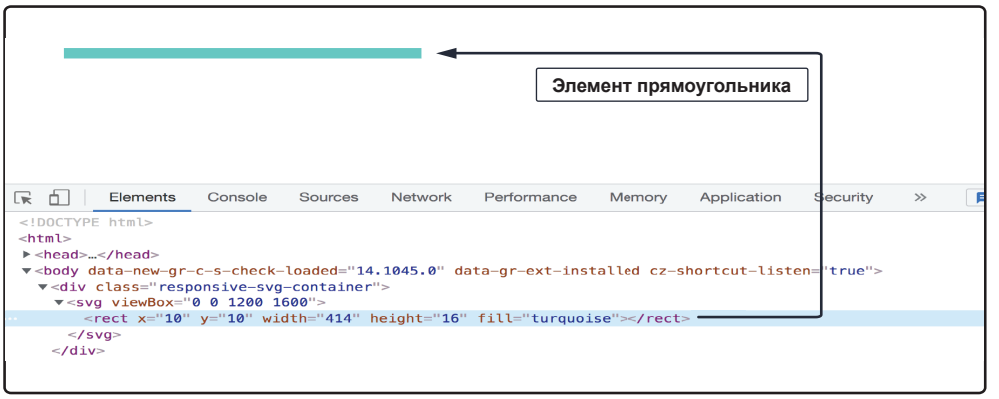


Рис. 2.15 Прямоугольник, добавленный в контейнер SVG

2.6 Установка и изменение стилей

Чтобы элементы визуализации имели привлекательный внешний вид, мы должны применить к ним стили оформления. Часто лучшим вариантом с точки зрения удобства сопровождения является традиционный подход с использованием таблиц стилей CSS. Но иногда удобнее напрямую установить и настроить атрибут `style` с помощью D3, особенно когда стили динамически изменяются вместе с данными.

D3 позволяет устанавливать и изменять атрибут `style` с помощью метода `selection.style()`. Этот метод принимает два параметра: имя свойства стиля и его значение, как показано на рис. 2.16.

```
selection.style("имя свойства стиля", "значение");
```

(строка)	(строка, число или функция доступа)
"fill"	"cyan"
"fill-opacity"	0.6
"stroke"	"plum"
"stroke-width"	2
"font-size"	"14px"

Рис. 2.16 Метод `style()`

Вернемся к нашему упражнению с гистограммой и в файле `main.js` применим метод `style()` к выборке с контейнером SVG, хранящейся в константе `svg`. Добавим, как показано ниже, рамку к контейнеру SVG. Вы можете назначить любое желаемое значение свойству стиля, а мы в этом примере решили сделать рамку черной с толщиной в 1 пиксель:

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600")
    .style("border", "1px solid black");
```

Рамка вокруг контейнера SVG поможет нам визуально обозначить рабочее пространство и понять, как работает метод `style()`.

Сохраните проект и посмотрите, как он выглядит в браузере. Найдите контейнер SVG в инспекторе. Вы должны увидеть свойство `border`, добавленное в атрибут `style`, как показано на рис. 2.17. Это означает, что метод `style()` благополучно внедрил то, что мы называем встроенными стилями.

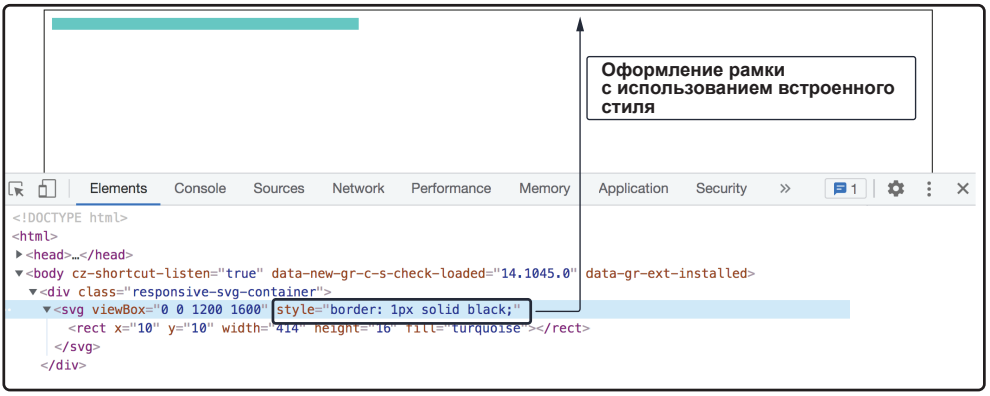


Рис. 2.17 Оформление рамки с помощью метода `style()`

При работе с элементами SVG некоторые визуальные свойства, такие как `fill` и `stroke`, можно определять через атрибуты или через встроенные стили. Не существует строгого правила, диктующего выбор между методами `attr()` и `style()` для применения таких свойств, но некоторые разработчики, стремясь быть последовательными, предпочитают задавать визуальные свойства через CSS или встроенные стили, а не через атрибуты. Это может пригодиться, например, когда мы решим сохранить каскад стилей простым в управлении, отделив код, который создает фигуры, от кода, определяющего их внешний вид. В этой книге для визуального оформления элементов SVG мы будем использовать оба метода, `attr()` и `style()`, а также внешние файлы CSS.

Проиллюстрируем сказанное на примере. В `main.js` добавьте вызов `style()` к выборке с прямоугольником и примените второе свойство `fill` с другим цветом. В следующем фрагменте мы используем цвет `"plum"`:

```
svg
  .append("rect")
    .attr("x", 10)
    .attr("y", 10)
    .attr("width", 414)
    .attr("height", 16)
    .attr("fill", "turquoise")
    .style("fill", "plum");
```

Теперь откройте файл `main.css` и добавьте третье свойство `fill` для применения к прямоугольнику. Здесь мы использовали цвет `"orange"`:

```
rect {
  fill: orange;
}
```

Сейчас у нашего прямоугольника есть три свойства `fill`, примененные разными способами: через атрибут мы определили цвет заливки `"turquoise"` (бирюзовый), через встроенный стиль – цвет `"plum"` (слива) и через внешнюю таблицу стилей CSS – цвет `"orange"` (оранжевый). Конечно, так мы никогда не будем делать в реальной жизни и сделали это здесь только ради демонстрации.

Сохраните проект и обратите внимание, что для заливки был выбран цвет, заданный через встроенное свойство стиля. На рис. 2.18 можно видеть, как применяется каскад стилей. Встроенный стиль переопределяет любые другие стили. Вторым по степени приоритета является стиль, определяемый внешней таблицей стилей CSS. А атрибут `fill` применяется, только если не определены два предыдущих свойства. Помня это правило, вы сможете разработать стратегию, соответствующую вашим привычкам, и вам не придется чесать голову, размышляя, почему один стиль виден на экране, а другой нет.

Теперь мы знаем, как произвести выборку, добавить элементы в DOM, а также настроить местоположение и оформление этих элементов. Но добавлять прямоугольники по одному, как мы только что сделали, очень неэффективно и утомительно. В следующей главе вы узнаете, как добавить все прямоугольники сразу, используя привязку данных. Но прежде удалим код, связанный с прямоугольником, из `main.js` и `main.css`. Файл `main.js` теперь содержит только код, представленный в листинге 2.2. Именно с него мы начнем строить столбчатую диаграмму в главе 3.

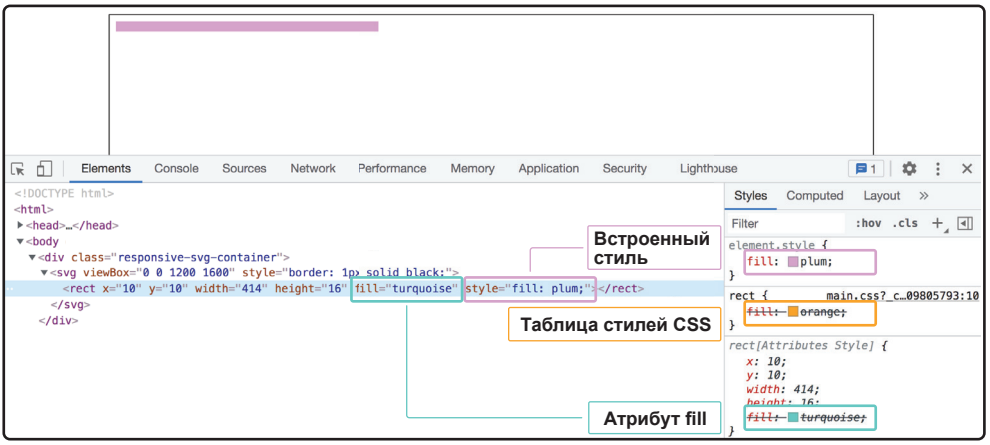


Рис. 2.18 Применение заливки через атрибут, через внешнюю таблицу стилей и через встроенный стиль

Листинг 2.2 Содержимое main.js в конце главы 2

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600")
    .style("border", "1px solid black");
```

Интервью с Надией Бремер

Надия Бремер (Nadieh Bremer) – художник и специалист по визуализации данных.

Расскажите коротко, как вы пришли в сферу визуализации данных и чем занимались до этого.

Я получила диплом астронома и поступила на работу в роли специалиста по обработке данных. Однако через несколько лет я поняла, что мне не хватает творческого аспекта. Когда я услышала название профессии «специалист по визуализации данных», то сразу поняла, что именно в ней я смогу найти то, чего мне так не хватает: творческую составляющую наряду с возможностью исследовать данные, интерпретировать их и экспериментировать с математикой.

Что привело вас к изучению D3, учитывая, что у вас не было опыта веб-разработки? С какими основными трудностями вы столкнулись и как их преодолели?

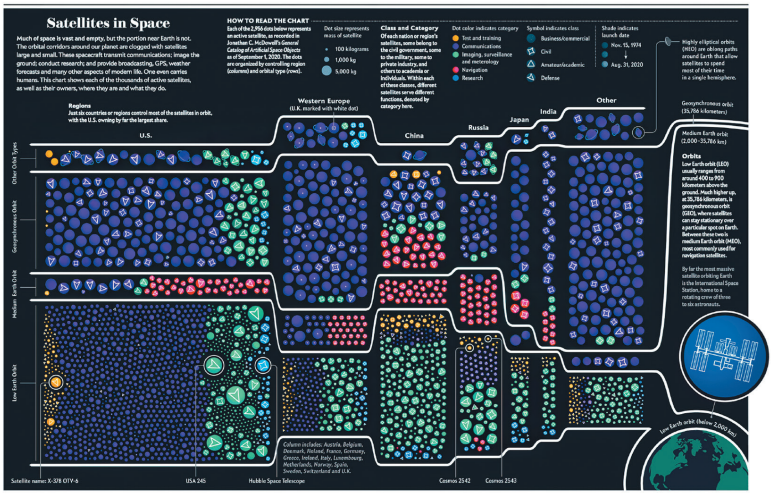
На одной из конференций (в 2013 году) я посетила семинар D3.js, где демонстрировались возможности D3, такие как интерактивные взаимодействия. На тот момент я не видела ничего подобного раньше и поняла, что хотела бы иметь возможность создавать похожие интерактивные визуализации данных. Я купила книгу *Interactive Data Visualization for the*

Web (O'Reilly, 2017) Скотта Мюррея (Scott Murray) и начала осваивать новую для меня область. Основной сложностью для меня было полное отсутствие знаний о HTML/CSS/JavaScript (хотя я знала, как программировать на R), поэтому, в довесок к D3, мне пришлось изучать и их. Будучи потрясающе невежественной, я не отличала уникальных особенностей D3 от чистого JavaScript, поэтому могла надолго застрять на чем-то незначительном, не понимая, как правильно сформулировать поисковый запрос, чтобы найти ответ в интернете. Мне очень помогли моя чрезвычайная настойчивость и уверенность в том, что я работаю над проектами, которые мне нравятся. Постепенно я начала понимать все больше и больше. Мне потребовался целый год, чтобы разобраться в шаблоне «enter-update-exit».

Используете ли вы другие инструменты визуализации, кроме D3? Какое место занимает D3 в вашем текущем рабочем процессе визуализации данных? Например, я вижу, что в своих проектах вы часто используете R.

На самом деле D3 – мой главный инструмент. Я использую его в 90 % своих проектов, когда приступаю к созданию финального визуального представления. Я также использую чистый JavaScript с некоторыми другими библиотеками, такими как Chroma.js и SimplexNoise. Но всегда начинаю с этапа анализа и подготовки данных в R и создаю множество простых диаграмм, чтобы получить понимание данных. Однако я никогда не использую их в окончательной визуализации.

Представленный проект: «Space Wars» (www.visualcinnamon.com/portfolio/space-wars/)



Визуализация спутников в космосе, выполненная Надией Бремер и опубликованная в журнале Scientific American

Расскажите о своем проекте «Space Wars», который вы создали для ноябрьского выпуска журнала Scientific American 2020 года.

Визуализация «Space Wars» показывает все спутники в околоземном пространстве, активно действовавшие на тот момент времени. Главной целью было показать, кому принадлежат спутники и на каких орбитах они находятся. Изображая каждый спутник как отдельный круг и группируя их по странам и орбитам, визуализация приводит также другую информацию о них, например вес, возраст, классификацию и многое другое.

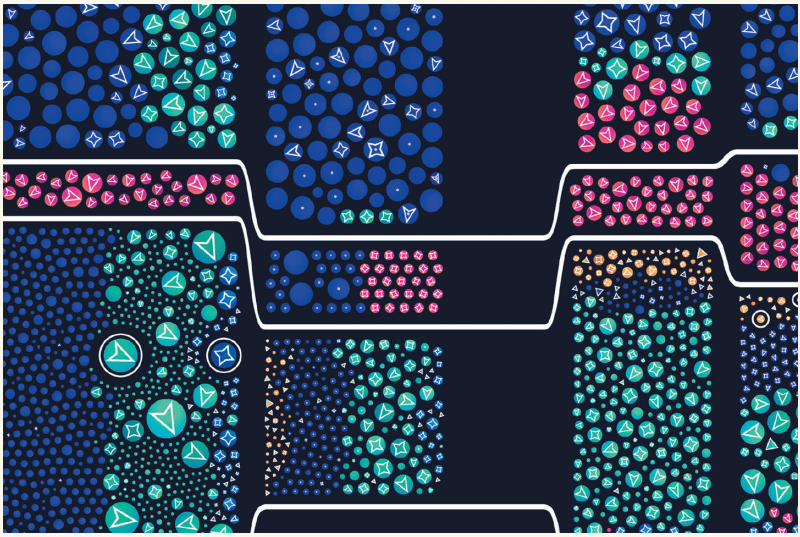
Расскажите, как создавался этот проект.

К моему счастью, данные были собраны экспертом по спутникам, поэтому мне не пришлось заниматься их обработкой, и нужно было лишь понять, что у меня есть. Я набросала несколько идей в блокноте. Я знала, что хочу отразить в визуализации распределение по странам и орбитам. Однако метаданных о каждом спутнике было так много, что поначалу я хотела просто показать каждый по отдельности. Я не знаю, как эта мысль пришла мне в голову, но в какой-то момент я нарисовала древовидную карту, заполненную кружками, изображающими спутники. После того как идея сформировалась в моей голове, осталась только техническая сторона дела: выяснить, как использовать кружки в древовидной карте и как сделать дизайн привлекательным. Сначала визуализация выглядела ужасно, но затем, потратив какое-то время на подбор цветов по мотивам некоторых моих любимых научно-фантастических фильмов, я получила то, что требовалось для визуального ряда.

Как у стороннего наблюдателя, у меня сложилось впечатление, что для создания визуализации вы объединили метод древовидной карты D3 с силовой компоновкой. Так ли это? Объединение методов из разных модулей для создания невиданных ранее макетов – одна из сторон D3, которые мне нравятся больше всего. И вы сами в своем курсе «Data Visualization: Customizing Charts for Beauty & Impact» предлагаете объединять разные типы диаграмм.

Вы не ошиблись! Я использовала метод древовидной карты, чтобы получить область (координаты левого верхнего угла, ширину и высоту), где должна разместить каждую группу спутников. Затем внутри каждого прямоугольника я запустила симуляцию силовой компоновки для размещения всех кругов так, чтобы они не перекрывались. Размер каждого квадрата для древовидной карты я определяла сложением площадей кругов (размеры которых определяются весом спутников) и умножением суммы на коэффициент преобразования, а иногда и вручную, прибегая к некоторому «магическому» числу, которое я определила вручную.

Да, возможность проявить творческий подход – это, безусловно, одна из моих любимых особенностей D3!



Круги, представляющие спутники, группируются по странам и типам орбит в древовидной карте. Затем их точное положение в визуализации рассчитывается с помощью метода силовой компоновки D3

Расскажите чуть больше об использовании D3 при создании статической визуализации для вывода на печать. Мне кажется, что мы недостаточно затронули эту сторону библиотеки!

Мне нравится создавать статические визуализации, и я почти всегда использую D3.js. Единственное, что важно для создания статической визуализации, – использование векторной графики SVG (вместо Canvas), чтобы верстальщики имели векторный файл, который можно масштабировать. Кроме того, в таких визуализациях сохраняется возможность правки шрифтов и заголовков. Большинство стилей, которые при этом применяются, сохраняются при переносе визуализации из браузера в векторный редактор (лучшим для этой цели оказался Illustrator).

Вашим проектом и книгой «Data Sketches» (A K Peters, 2021), созданными совместно с Ширли Ву (Shirley Wu), вы вдохновили многих из нас. Какой совет вы бы дали тем, кто начинает работать в этой области?

Мне приятно слышать, что книга «Data Sketches» кого-то вдохновила. Странно вспоминать, как мы с Ширли начинали работу над ней, ожидая, что это будет малозаметный проект, который оценим только мы и несколько близких друзей.

Я могу дать два совета. Во-первых: творите, создавайте как можно больше визуализаций. С каждым новым проектом вы будете узнавать что-то новое (иногда даже не осознавая этого), столкнетесь с новыми подводными камнями и освоите новые способы их преодоления. Когда я оглядываюсь на пройденный мною путь и визуализации, созданные

на этом пути, мне становится совершенно ясно, что я стала более уверенной в выборе решений и более творческой, и это сильно повлияло на все, что я делаю.

Во-вторых: пробуйте работать с данными и темами, которые вас действительно интересуют, даже если для сбора данных требуется ручной труд. Интерес к теме вдохновляет на продолжение работы, даже если вы застряли (запутались в каких-то технических или творческих аспектах). Вы можете видеть это в книге «Data Sketches». Мы никогда не выбирали легкий путь с точки зрения данных, но всегда старались следовать за нашими интересами и почти всегда обнаруживали необходимые нам данные.

Итоги

- Для запуска проектов D3 требуется веб-сервер. Самый простой способ получить доступ к такому веб-серверу в локальной среде разработки – использовать расширение Live Server для VS Code.
- Загрузить библиотеку D3 в проект можно двумя способами: добавить в `index.html` тег `script`, ссылающийся на библиотеку, или установить в виде модуля NPM. В первых главах этой книги для простоты мы используем подход с тегом `script`.
- Библиотеку D3 можно загрузить целиком или только необходимые модули. В последнем случае можно несколько повысить производительность проекта.
- При загрузке файлов и библиотек с использованием тега `script` порядок загрузки будет совпадать с порядком следования тегов. Это означает, что тег `script`, ссылающийся на библиотеку D3, должен располагаться перед тегами `script`, загружающими файлы JavaScript, которые используют D3. В противном случае браузер не будет иметь доступа к вызываемым методам D3 и прекратит выполнение сценариев.
- D3 предоставляет методы `d3.select()` и `d3.selectAll()`, позволяющие выбирать элементы из DOM. Первый метод возвращает только первый найденный элемент, а второй – все элементы, соответствующие указанному селектору.
- Селекторы, передаваемые в вызов `select()` и `selectAll()`, идентичны селекторам в таблицах стилей CSS. В них можно использовать имена тегов и классов, идентификаторы или их комбинации.
- Метод `append()` позволяет добавить элемент в конец выбранного элемента.

- Метод `attr()` позволяет добавлять или изменять атрибуты элемента. Он принимает два аргумента: имя атрибута и его значение.
- Метод `style()` позволяет устанавливать и изменять атрибут стиля элементов DOM. Он тоже принимает два аргумента: имя свойства стиля и его значение.
- С помощью метода `style()` можно применять встроенные стили, которые имеют более высокий приоритет перед стилями, указанными в таблицах стилей CSS, добавлять стилевые атрибуты.

3 Работа с данными

В этой главе рассматриваются:

- распознавание типа данных;
- загрузка, форматирование и оценка данных;
- привязка данных к элементам DOM;
- использование шкал для преобразования данных в визуальные атрибуты;
- добавление текстовых меток в диаграммы.

Общим для всех визуализаций является наличие базовых данных. Мы как разработчики визуализаций сталкиваемся с различными типами и наборами данных, которые необходимо представить в наглядной форме. В этой главе мы обсудим типичный процесс работы с данными, применимый к большинству проектов D3. Основная стратегия, изображенная на рис. 3.1, начинается с поиска данных для проекта. Данные могут содержать значения различных типов, номинальные или порядковые, и могут поступать в различных форматах, таких как файлы CSV или JSON. Обычно на этом этапе требуется проделать много работы по подготовке и очистке данных, но мы не будем рассматривать данный аспект.



Рис. 3.1 Рабочий процесс обработки данных в D3

После формирования набора мы можем использовать D3 для загрузки, форматирования и оценки составляющих его данных. После этого можно приступать к созданию визуальных элементов, обычно состоящих из фигур SVG. Этот процесс называется связыванием данных, и мы будем использовать его для создания прямоугольников в столбчатой диаграмме, которую мы начали строить в главе 2.

Значения, содержащиеся в наборе данных, не всегда можно использовать напрямую. Числа могут быть слишком большими, чтобы играть роль размеров визуальных элементов, или, может быть, нам захочется представить определенные значения цветами. В таких случаях нам на выручку приходят шкалы D3. Далее мы обсудим различные типы шкал и способы их применения. Потом воспользуемся линейными и интервальными шкалами для позиционирования и изменения размера прямоугольников в нашей гистограмме (http://mng.bz/mjor).

На рис. 3.2 показана упрощенная версия процесса обработки данных, этапы которого мы будем рассматривать в этой главе.

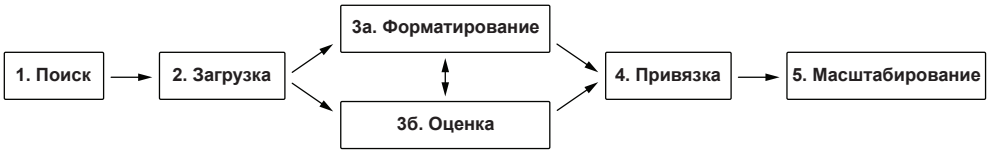


Рис. 3.2 Упрощенный рабочий процесс создания диаграммы в D3

3.1 Изучение данных

Первый шаг в рабочем процессе обработки данных – поиск или формирование набора данных (рис. 3.3). Прежде чем погрузиться в методы работы с данными, давайте кратко обсудим различные типы и форматы данных, с которыми вы столкнетесь как разработчик D3. Этот небольшой теоретический экскурс поможет вам получить представление об особенностях данных, что совершенно необходимо для выбора подходящей архитектуры визуализации.



Рис. 3.3 Первый шаг рабочего процесса в D3 – поиск или формирование набора данных. Данные бывают разных типов и форматов

3.1.1 Поиск данных

Иногда вам придется формировать свои наборы данных, но чаще вы будете использовать уже готовые наборы. В интернете есть масса ресурсов, где можно найти интересные и общедоступные наборы, вот некоторые из них:

- Data.world (https://data.world/);
- Kaggle (www.kaggle.com/);
- World Bank Open Data (https://data.worldbank.org/);
- Our World in Data (https://ourworldindata.org/);
- EarthData (www.earthdata.nasa.gov/);
- EU Open Data Portal (https://data.europa.eu/);
- Data.gov (https://data.gov/);
- openAfrica (https://africaopendata.org/);
- Dataportal.asia (https://dataportal.asia/).

3.1.2 Типы данных

При построении визуализаций мы работаем с двумя основными типами данных: количественными и качественными. Количественные данные – это числовая информация, такая как время, вес или объем валового внутреннего продукта (ВВП) страны. Количественные данные, как показано на рис. 3.4, могут быть дискретными или непрерывными. Дискретные данные состоят из натуральных чисел, также называемых целыми числами, которые не имеют дробной части. Например, число сотрудников в компании может быть равно 16, но

никогда не будет равно 16,3. С другой стороны, непрерывные данные можно делить на более мелкие единицы, и они все равно будут иметь смысл. Типичным примером непрерывных данных может служить температура. Мы можем сказать, что сегодня температура воздуха 17 °C, но ее можно измерить с большей точностью и узнать, что на самом деле она составляет 16,8 °C. Как правило, непрерывные данные можно измерить с помощью прибора, тогда как дискретные – только подсчитать, но не измерить.

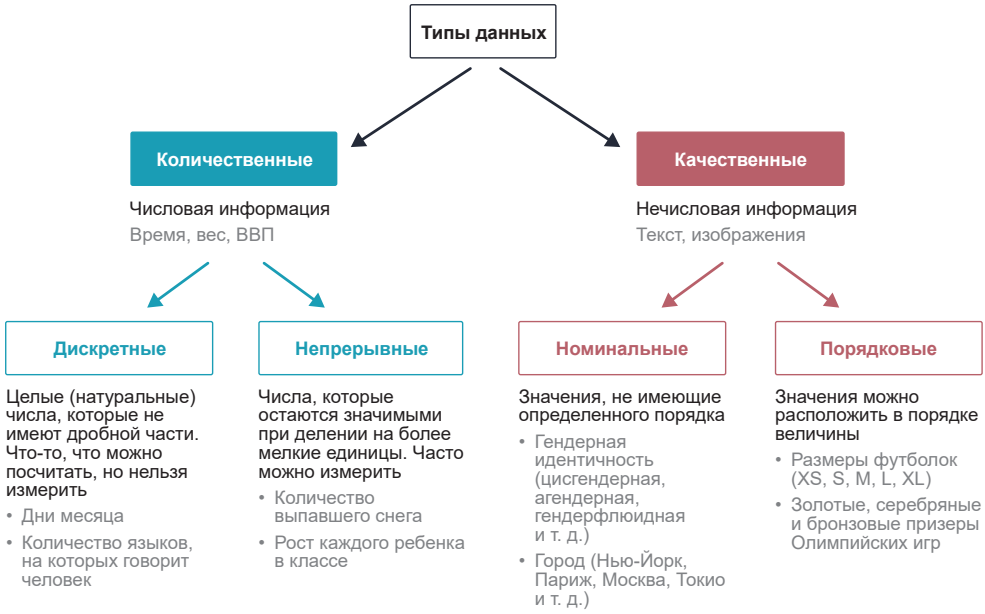


Рис. 3.4 Классификация типов данных

Качественные данные, с другой стороны, состоят из нечисловой информации, такой как текст. Они могут быть номинальными или порядковыми. Номинальные значения не имеют определенного порядка, например метки гендерной идентичности или названия городов. Порядковые значения, напротив, можно классифицировать по порядку величины. Если взять в качестве примера размеры футболок, то мы обычно перечисляем их в порядке возрастания размера (XS, S, M, L, XL).

Типы данных влияют на выбор вида визуализации для представления данных. Например, линейные диаграммы отлично подходят для непрерывных данных, но не подходят для дискретных. Номинальные значения можно представить набором цветов, соответствующих категориям, тогда как для порядковых значений можно выбрать цветовую палитру.

При выборе шкалы D3 также учитываются типы данных. Линейные шкалы используются с количественными данными, а для ка-

чественных в D3 имеются специальные шкалы. Более подробно о шкалах мы поговорим в разделе 3.4.

3.1.3 Форматы и структуры данных

Для разных целей данные можно отформатировать по-разному. Тем не менее они, как правило, относятся к нескольким хорошо знакомым структурам: табличные данные, объекты JavaScript (JSON), иерархические данные, сети, географические данные и неструктурированные данные. Эти структуры не являются взаимоисключающими.

Табличные данные

Табличные данные представлены в виде строк и столбцов и обычно находятся в электронной таблице или в таблице в базе данных. Табличные данные разделяются определенным символом, который называют разделителем. Наиболее типичным примером табличных данных являются файлы значений, разделенных запятыми (Comma-Separated Values, CSV), где разделителем является запятая. Также можем встретить файлы значений, разделенных табуляцией (Tab-Delimited Values, TSV), или любые другие файлы значений, разделенных разделителем (Delimiter-Separated Values, DSV), в которых используются определенные разделители, такие как вертикальная черта или точка с запятой.

Возьмем для примера набор данных о наших вымышленных сотрудниках, обсуждавшийся в главе 1. Если сохранить эти данные в файле CSV, TSV или DSV, то значения были бы соответственно разделены запятой, табуляцией или другим разделителем, как показано в табл. 3.1. В табличных наборах данных в первой строке обычно перечисляются заголовки столбцов, а каждая строка (или запись) занимает отдельную строку.

Таблица 3.1. Данные в форматах CSV, TSV и DSV

CSV: значения, разделенные запятыми	TSV: значения, разделенные табуляцией	DSV: значения, разделенные разделителем (вертикальной чертой)
`id,name,works_with_d3`	`id name works_with_d3`	`id|name|works_with_d3`
1,Зоя,false	1 Зоя false	1|Зоя|false
2,Джеймс,true	2 Джеймс true	2|Джеймс|true
3,Алиса,true	3 Алиса true	3|Алиса|true
4,Хуберт,false	4 Хуберт false	4|Хуберт|falseя

D3 предлагает три функции для загрузки табличных данных: `d3.csv()`, `d3.tsv()` и `d3.dsv()`. Единственное различие между ними: `d3.csv()` загружает файлы значений, разделенных запятыми,

`d3.tsv()` – файлы значений, разделенных табуляцией, а `d3.dsv()` позволяет указать, какой разделитель используется. В этой главе и далее в книге мы будем использовать `d3.csv()`.

JSON

Файлы JSON (JavaScript Object Notation – форма записи объектов JavaScript) часто используются для хранения простых структур данных. Разработчики регулярно используют их, особенно при извлечении информации из конечных точек API, которые возвращают данные только в формате JSON.

Если сохранить данные из табл. 3.1 в формате JSON, то они выглядели бы как следующий массив объектов. Это не самый компактный вариант, и все же он имеет существенное преимущество, обеспечивая простой доступ к парам ключ–значение с применением точечной нотации JavaScript, обсуждаемой в главе 1:

```
[
  {
    "id": 1,
    "name": "Зоя",
    "position": "Аналитик данных",
    "works_with_d3": false
  },
  {
    "id": 2,
    "name": "Джеймс",
    "position": "Разработчик веб-интерфейсов",
    "works_with_d3": true
  },
  {
    "id": 3,
    "name": "Алиса",
    "position": "Разработчик полного профиля",
    "works_with_d3": true
  },
  {
    "id": 4,
    "name": "Хуберт",
    "position": "Дизайнер",
    "works_with_d3": false
  }
]
```

Для загрузки файлов JSON библиотека D3 предлагает функцию `d3.json()`. Но независимо от функции загрузки табличные данные преобразуются в массив объектов, в результате чего получается структура, подобная JSON.

Иерархические данные

Иерархические данные, в которых одни объекты существуют как вложенные внутри других, – обычное явление. Многие из наиболее интуитивно понятных макетов в D3 основаны на иерархических данных, которые могут быть представлены в виде деревьев, как на рис. 3.5, или упакованы в круги либо прямоугольники. Загрузка иерархических наборов данных требует писать дополнительный код, но гибкость этого представления стоит потраченных усилий. Поближе с иерархическими данными мы познакомимся в главе 11.

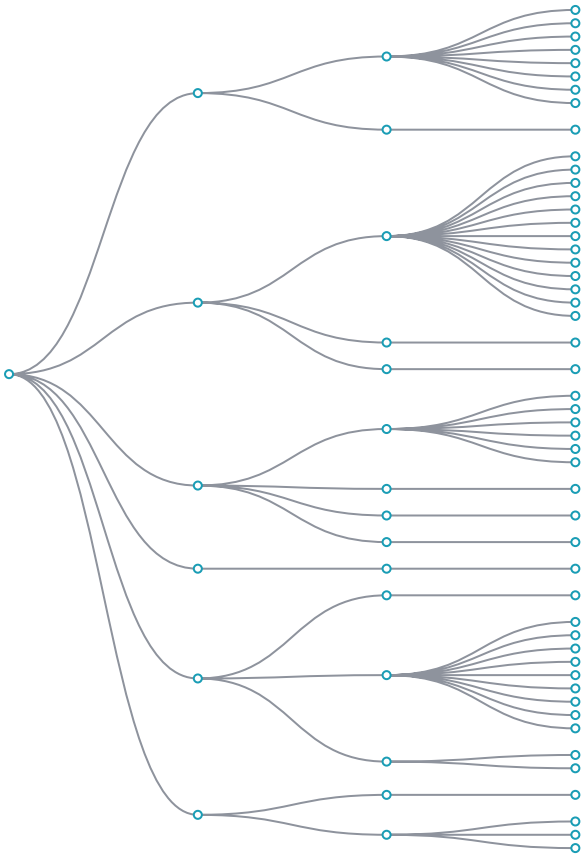


Рис. 3.5 Иерархические данные представляют отношения родитель–потомок между объектами. Каждый объект обычно имеет массив дочерних объектов, связанных как на этой дендрограмме. Обратите внимание, что каждый объект может иметь только одного родителя

Сеть

Сети можно увидеть повсюду: связи в социальной сети, транспортные сети или сетевые блок-схемы. Они являются мощным методом

представления сложных систем. Сетям часто придается вид диаграмм узлов и связей, как на рис. 3.6. Как и географические данные, сетевые данные имеют много стандартов, но в этой книге мы сосредоточимся только на списках узлов и ребер. Сетевые данные также легко исследовать с помощью бесплатного инструмента, такого как Gephi (доступен по адресу https://gephi.org). Подробнее сетевые данные и их стандарты мы рассмотрим в главе 12.



Рис. 3.6 Сетевые данные состоят из объектов и связей между ними. Объекты обычно называются узлами или вершинами, а связи – ребрами или соединениями. Сети часто визуализируются с использованием алгоритмов уравновешивания сил, как в этом примере, организующих сеть так, чтобы уравновесить силы, представленные соединениями между узлами

Географические данные

Географические данные описывают местоположение точек или форм и используются для создания разнообразных онлайн-карт, таких как карта мира на рис. 3.7 или карты в галерее Observable (https://observablehq.com/@d3/gallery#maps). Благодаря невероятной популярности веб-картографии нам доступно огромное количество открытых геоданных для любого проекта. Есть несколько стандартов представления географических данных, но в этой книге основное внимание уделяется двум из них: GeoJSON и TopoJSON. Геоданные могут иметь множество форм, а легкодоступные инструменты географических информационных систем (Geographic Information System, GIS), такие как PostGIS (https://postgis.net), позволяют преобразовывать данные в формат GIS, пригодный для публикации в интернете. Мы детально исследуем географические данные в главе 13.

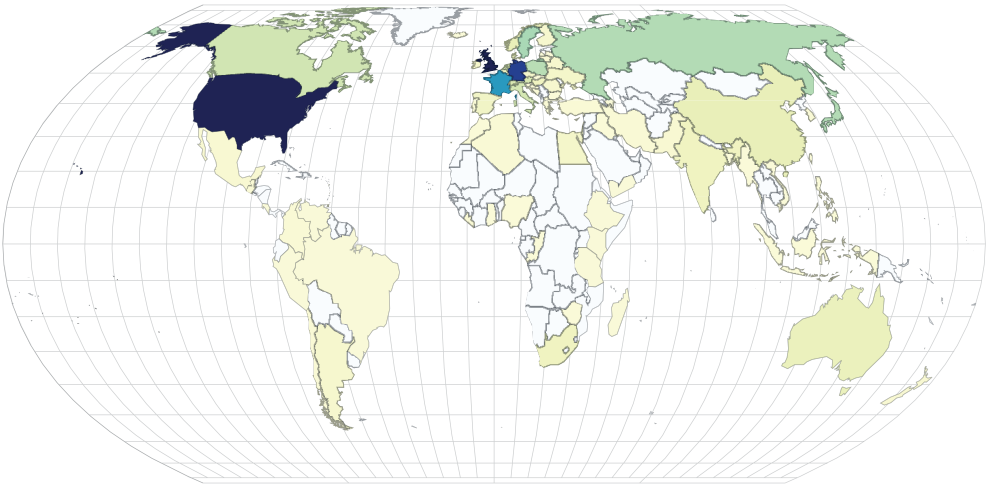


Рис. 3.7 Географические данные хранят пространственную геометрию объектов. Каждая страна на этом изображении представлена как отдельный объект с массивом значений, определяющих ее форму. Географические данные также могут состоять из точек (например, обозначающих города) или линий (например, для представления дорог)

Неструктурированные данные

По мере знакомства с отраслью визуализации вы обнаружите, что все вокруг является данными, включая изображения, блоки текста и разметку веб-сайта. Для представления данных визуализации обычно используют формы разного цвета и размера, но иногда лучший способ передачи информации – простой повествовательный текст, изображение или видео. Если вы разрабатываете приложения для тех, кто должен понимать сложные системы, но считаете, что манипуляция текстом или изображениями не связана с представлением числовых или категориальных данных в виде форм, то вы фактически ограничиваете свои возможности общения. Макеты и форматирование, используемые при работе с текстом и изображениями, характерные для классических веб-публикаций, возможны и в D3.

3.2 Подготовка данных

Получив готовый набор данных, мы загружаем его в проект, согласно шагу 2 на рис. 3.8, и приступаем к форматированию и оценке различных аспектов данных. В этом разделе мы обсудим методы D3, используемые для решения этих задач и подготовки данных для нашей столбчатой диаграммы.



Рис. 3.8 Второй шаг рабочего процесса в D3 – загрузка набора в проект с помощью одной из функций

3.2.1 Загрузка набора данных в проект D3

D3 имеет ряд удобных функций для загрузки наборов данных в проекты. Выбор функции напрямую зависит от формата набора данных. Например, файлы CSV загружаются с помощью `d3.csv()`, а файлы JSON – с помощью `d3.json()`. Путь к файлу данных передается функциям в первом аргументе. D3 также имеет функции для загрузки текстовых и даже XML-файлов. Все они определены в модуле d3-fetch (https://d3js.org/d3-fetch).

> **ПРИМЕЧАНИЕ** Файлы с примерами кода для главы 3 можно найти в репозитории книги на GitHub (http://mng.bz/Y7Po). Примеры для разных разделов находятся в соответствующих папках. Код для раздела 3.2 можно найти в папке с именем, начинающимся с номера `3.2`, за которым следует название темы, например `3.2-Preparing_data` (папка с примерами для раздела «3.2. Подготовка данных»). Каждая папка содержит вложенные папки `start` и `end`. Используйте папку `start`, если хотите начать с нуля, получив минимальный код. Если растеряетесь и захотите посмотреть решение, то вы найдете его в папке `end`. Работая с главой, открывайте в редакторе только одну папку, `start` или `end`. Если открыть сразу обе и использовать расширение Live Server для обслуживания проекта, то некоторые пути не будут работать так, как ожидалось, особенно это касается загрузки набора данных в проект.

А теперь вернемся к упражнению с гистограммой, начатому в главе 2, и загрузим пример набора данных из результатов исследования «2021 Data Visualization State of the Industry Survey» (www.datavisualizationsociety.org/survey). Набор имеет формат CSV и находится в папке `/data`. Содержимое файла `data.csv` показано в листинге 3.1. В первой строке перечислены заголовки столбцов, `technology` и `count`, которые представляют инструменты визуализации данных, участвовавшие в исследовании, от `ArcGIS` до `P5`, и количество специалистов по визуализации, выбравших каждый инструмент (414 выбрали D3.js и 530 – Python).

Листинг 3.1 Инструменты, наиболее часто используемые специалистами по работе с данными (data.csv)

```
technology,count
ArcGIS,147
D3.js,414
Angular,20
Datawrapper,171
Excel,1078
Flourish,198
ggplot2,435
Gephi,71
Google Data Studio,176
Highcharts,58
Illustrator,426
Java,29
Leaflet,134
Mapbox,167
kepler.gl,24
Observable,157
Plotly,223
Power BI,460
PowerPoint,681
Python,530
QGIS,193
Qlik,61
R,561
React,145
Tableau,852
Vega,48
Vue,51
Web Components,79
WebGL,65
Pen & paper,522
Physical materials,69
Canvas,121
P5/Processing,55
```

Поскольку набор данных хранится в CSV-файле, мы можем загрузить его в проект с помощью функции `d3.csv()`, передав ей путь к файлу в первом параметре. Так как файл данных находится в папке `/data`, относительный путь к нему из `main.js` выглядит так: `"../data/data.csv"`. Здесь две точки (`".."`) обозначают каталог уровнем выше текущего, т. е. корневой каталог проекта, а последующий путь ведет в папку `/data` к файлу `data.csv`:

```
d3.csv("../data/data.csv");
```

Этот вызов завершает шаг 2 рабочего процесса обработки данных, но теперь нам нужно понять, как получить доступ к данным для

форматирования и оценки. Важно знать, что загрузка данных – это *асинхронный процесс*. Асинхронные операции – это запросы, результат которых недоступен немедленно. Нам нужно дождаться, пока D3 извлечет данные, и только потом читать или обрабатывать их. Мы можем без опаски манипулировать загружаемыми данными в функции обратного вызова, которая передается в `d3.csv()`, или воспользоваться JavaScript-объектом `Promise`, чтобы дождаться окончания загрузки.

3.2.2 Форматирование набора данных

В этом разделе мы обсудим приемы форматирования данных с помощью D3. Это третий шаг на рис. 3.9.



Рис. 3.9 Третий шаг рабочего процесса в D3 предполагает обязательное форматирование данных с целью их подготовки к использованию в визуализации

Функцию обратного вызова, что передается в `d3.csv()`, также называют функцией преобразования записей, потому что она получает доступ к данным построчно. В следующем фрагменте первый аргумент `d3.csv()` – это путь к данным, а второй – функция обратного вызова, которая выводит данные в консоль. Скопируйте этот фрагмент в `main.js` и сохраните проект:

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600")
    .style("border", "1px solid black");

d3.csv("../data/data.csv", d => {
  console.log(d);
});
```

Откройте панель инспектора в браузере и перейдите на вкладку **Console** (Консоль). Вы увидите (рис. 3.10), что данные выводятся по одной строке за раз и каждая строка выглядит как объект JavaScript, с полями `technology` и `count`.

Обратите внимание, что значения из столбца `count` извлечены как строки, а не числа. Это распространенная проблема при импорте данных, связанная с преобразованием набора данных из формата CSV в формат JSON. Поскольку функция обратного вызова имеет доступ к данным по одной строке за раз, ее с успехом можно использо-

вать для преобразования счетчиков обратно в числа. Это гарантирует, что значения счетчиков будут готовы к использованию в нашей визуализации.

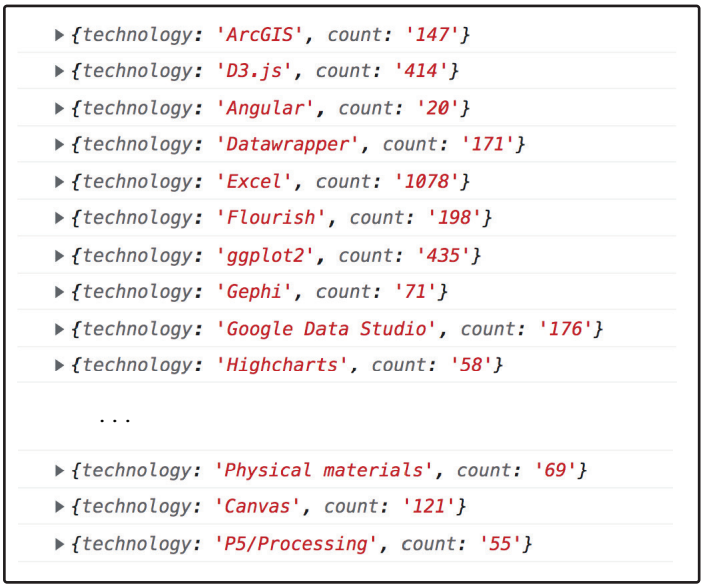


Рис. 3.10 Извлеченные данные, записанные в консоль функцией обратного вызова

В следующем фрагменте вместо вывода каждой строки данных в консоль мы возвращаем объект, содержащий пары ключ–значение с технологиями и счетчиками. Значения извлекаются из параметра `d` с использованием точечной нотации. В процессе загрузки мы снова и снова получаем параметр `d` с объектами, которые ранее выводились в консоль (рис. 3.10), и извлекаем технологию обращением к `d.technology` и счетчик – обращением к `d.count`, формируя пары ключ–значение. Оператор `+` перед `d.count` преобразует это значение в число:

```
d3.csv("../data/data.csv", d => {
  return {
    technology: d.technology,
    count: +d.count
  };
});
```

Важно понимать, что пары ключ–значение, возвращаемые функцией обратного вызова, – это единственное, что будет доступно после загрузки набора данных. Эта стратегия помогает эффективно избавиться от названий столбцов в исходном наборе данных, которые

нам не нужны. Но если набор данных содержит много столбцов и их все нужно сохранить, то возврат ключей и значений по одному может оказаться избыточным. В таком случае можно ничего не делать в функции обратного вызова и произвести форматирование после того, как D3 вернет полный набор данных. Как получить доступ ко всему набору данных сразу, мы расскажем в следующем разделе.

3.2.3 Оценка набора данных

Третий шаг рабочего процесса в D3 включает необязательную оценку данных (рис. 3.11). Прием построчного извлечения данных может оказаться полезным, но также было бы желательно иметь доступ к набору данных в целом. В этом нам могут помочь JavaScript-объекты `Promise`. Объект `Promise` – это результат выполнения асинхронной операции, такой как `d3.csv()`. Самый простой способ извлечь данные из `Promise` – использовать метод `then()`.

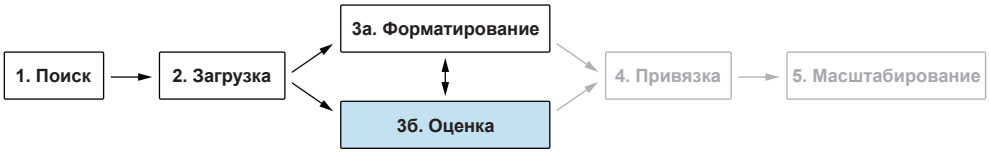


Рис. 3.11 Вторая часть третьего шага – оценка и исследование данных

Следующий фрагмент демонстрирует применение метода `then()` к результату, возвращаемому функцией `d3.csv()`. После полной загрузки данных объект `Promise` активируется и полный набор данных становится доступен в функции обратного вызова, переданной методу `then()`. Добавьте функцию, которая выводит набор данных в консоль, и сохраните проект:

```
d3.csv("../data/data.csv", d => {
  return {
    technology: d.technology,
    count: +d.count
  };
}).then(data => {
  console.log(data);
});
```

Вывод в консоли показывает, что набор данных был преобразован в массив объектов, каждый из которых представляет строку из исходного набора данных CSV. Итак, мы выяснили, что D3 обеспечивает возможность итераций по данным, очень удобную для разработки визуализаций. Мы также можем видеть (рис. 3.12), что значения счетчиков правильно преобразованы в числа. Если посмотреть

на последний элемент в массиве данных, то можно увидеть, что D3 поместила в него заголовки столбцов из файла CSV. Мы не будем использовать их в нашей столбчатой диаграмме, но иногда этот массив может очень даже пригодиться.

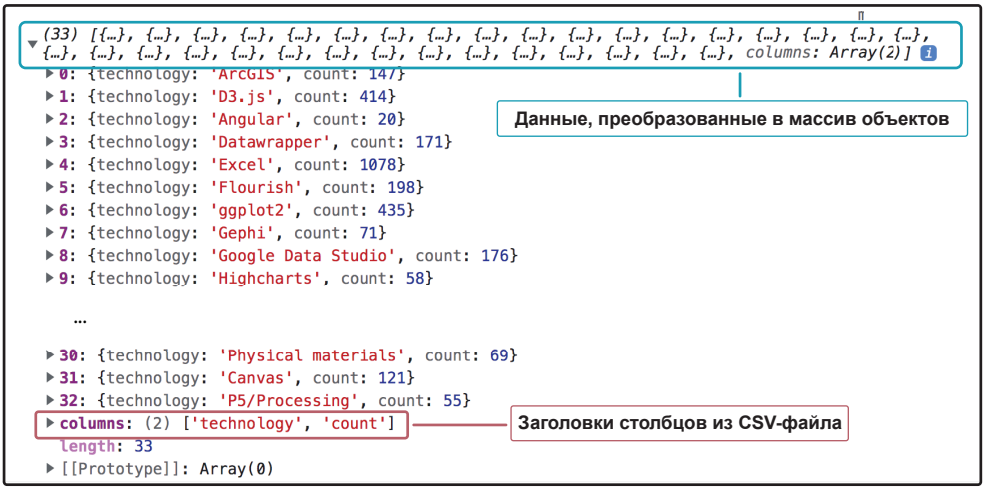


Рис. 3.12 Набор данных в виде массива объектов в консоли

На шаге 3а рабочего процесса в D3 мы отформатировали полученные данные и теперь можем исследовать эти данные и вычислять некоторые статистические оценки с помощью D3. Вычисление оценок может дать понимание некоторых дополнительных аспектов, необходимое для более осознанного погружения в работу над созданием визуализации.

В целом нет строгих правил, определяющих, где должны проводиться вычисления, однако отличным местом для первичного исследования данных является метод `then()`. Первое, что можно узнать, – количество технологий в наборе. Для этого можем прочитать свойство `length` массива данных. Если в нашем примере вывести значение свойства `length`, то мы увидим число 33, т. е. наша столбчатая диаграмма будет иметь 33 прямоугольника:

```
d3.csv("../data/data.csv", d => {
  ...

}).then(data => {

  console.log(data.length);    // => 33

});
```

Количество записей в наборе данных

Нам также интересно знать, какая технология пользуется наибольшей популярностью по данным нашего опроса и сколько спе-

циалистов заявили, что используют ее регулярно. Не менее интересно узнать, какая технология наименее популярная. Получить эти значения можно с помощью методов `d3.max()` и `d3.min()`. Как показано в следующем фрагменте, эти методы принимают два параметра. Первый – итерируемый объект, из которого нужно получить максимальное или минимальное значение (это данные, возвращаемые объектом `Promise`). Второй параметр – функция доступа, которая определяет, значения каких атрибутов в объектах данных будут сравниваться, – в данном случае `count`.

Если вывести максимальное и минимальное значения в консоль (все так же внутри метода `then()` объекта `Promise`), то мы получим `1078` и `20` соответственно:

```
d3.max(data, d => d.count)      // => 1078
d3.min(data, d => d.count)      // => 20
d3.extent(data, d => d.count)   // => [20, 1078]
```

> **ПРИМЕЧАНИЕ** Аналогично можно использовать метод `d3.extent()`, который принимает те же параметры и возвращает массив с минимальным и максимальным значениями.

Знание максимальных и минимальных значений помогает примерно прикинуть длину столбцов в диаграмме и понять, насколько легко будет отобразить на экране разницу между максимальным и минимальным значениями.

В столбчатых диаграммах общепринято сортировать данные в порядке убывания. Это облегчает их чтение и позволяет быстро узнать, какие технологии более или менее популярны. Отсортировать данные можно с помощью JavaScript-метода `sort()`. Он принимает функцию сравнения в качестве аргумента, как показано в следующем фрагменте, которая в данном случае сравнивает значения `count` двух технологий, представленных параметрами `a` и `b`. Если количество `b` больше количества `a`, то `b` должно находиться перед `a` в отсортированном массиве и т. д.:

```
data.sort((a, b) => b.count - a.count);
```

Отсортировать данные можно все в том же методе `then()`. Если вывести отсортированный массив в консоль, то вы увидите, что Excel находится в начале списка технологий с числом 1078, за ним следует Tableau с числом 852. А последней технологией в столбчатой диаграмме будет Angular с числом 20 (рис. 3.13).

Модуль d3-array (https://d3js.org/d3-array) содержит много других методов для оценки и преобразования данных, часть из которых мы рассмотрим в этой книге. Но наиболее часто, пожалуй, вы будете использовать `d3.max()`, `d3.min()` и `d3.extent()`.

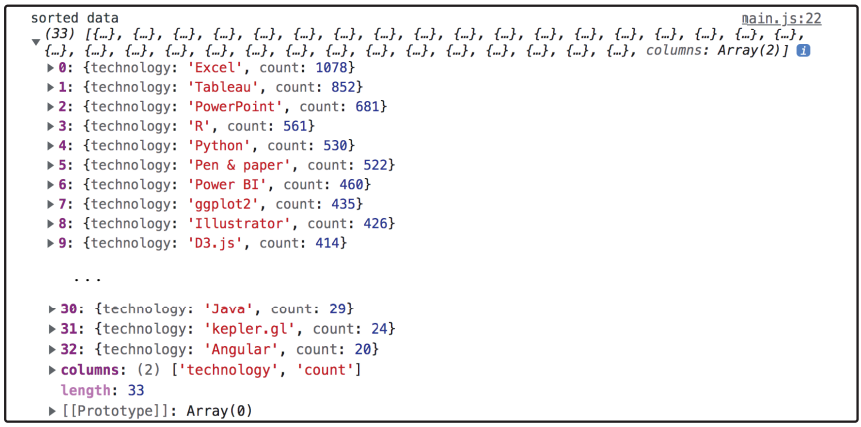


Рис. 3.13 Набор данных, отсортированный в порядке убывания популярности технологий

По завершении загрузки, преобразования и оценки данных общепринято передавать набор данных в другую функцию, которая занимается построением визуализации. В листинге 3.2 представлено текущее состояние `main.js`, где можно видеть, что в конце метода `then()` мы передаем данные в функцию `createViz()`. В следующем разделе мы приступим к созданию этой функции.

Листинг 3.2 Загрузка, преобразование и оценка данных (main.js)

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
    .attr("viewBox", "0 0 1200 1600")
    .style("border", "1px solid black");

d3.csv("../data/data.csv", d => {

  return {
    technology: d.technology,
    count: +d.count
  };

}).then(data => {

  console.log(data.length);                    // => 33
  console.log(d3.max(data, d => d.count));     // => 1078
  console.log(d3.min(data, d => d.count));     // => 20
  console.log(d3.extent(data, d => d.count)); // => [20, 1078]

  data.sort((a, b) => b.count - a.count);
  createViz(data);

});

const createViz = (data) => {};
```

Добавление контейнера SVG

Загрузка набора данных

Форматирование данных

Оценка набора данных

Сортировка в порядке убывания

Передача данных в другую функцию

Функция, в которой мы построим столбчатую диаграмму

Прежде чем завершить этот раздел, давайте вспомним этапы пройденного пути: загрузку данных, преобразование строк и приемы работы с объектом Promise (рис 3.14):

1. Загрузка данных была выполнена с помощью функции `d3.csv()`.
2. Форматирование производилось с применением функции преобразования записей.
3. Для получения доступа к полному набору данных после загрузки в цепочку вызовов был добавлен метод `then()`. Этот метод – отличное место для оценки данных и выполнения любых операций, применяемых ко всему набору данных.
4. Данные переданы другой функции, которая будет заниматься построением визуализации.

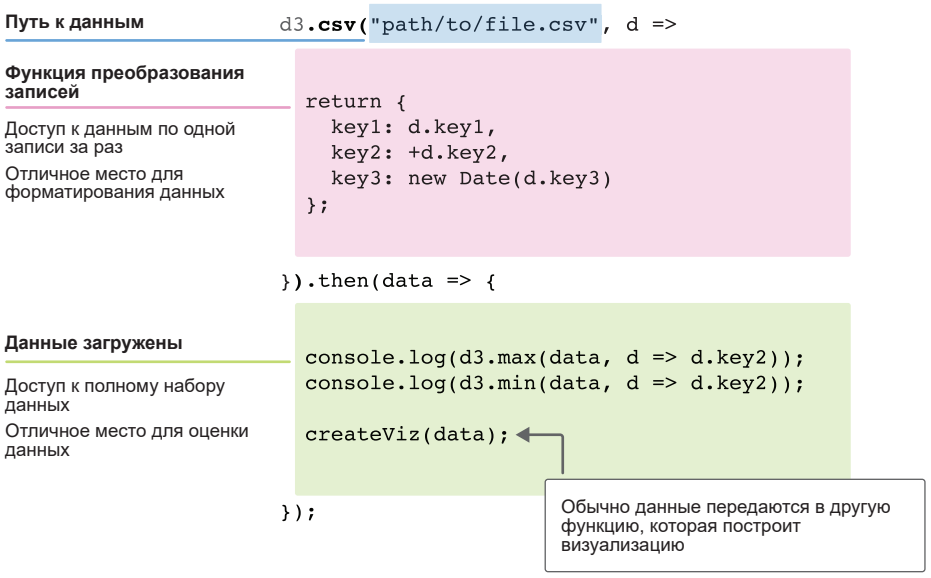


Рис. 3.14 Как и где загружать, преобразовывать и оценивать данные в D3

3.3 Привязка данных к элементам DOM

Теперь вы готовы познакомиться с одной из самых захватывающих возможностей D3: привязкой данных. Привязка данных позволяет связывать отдельные фрагменты данных с элементами DOM. Например, мы свяжем каждый SVG-элемент `rect` в нашей столбчатой диаграмме с технологией и соответствующим ей значением счетчика. На этапе привязки данных (четвертый этап на рис. 3.15) визуализация начинает оживать – и это всегда радостный момент для разработчика визуализации!

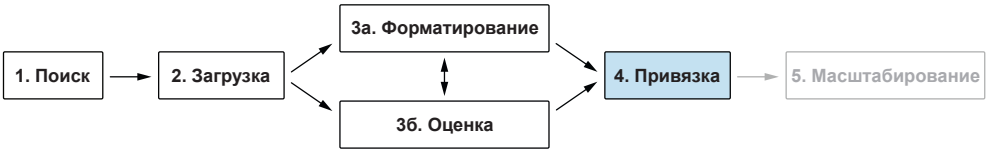


Рис. 3.15 Четвертый шаг рабочего процесса в D3 включает создание элементов DOM, которые станут ядром визуализации, и привязку к ним данных

Чтобы привязать данные, достаточно просто использовать шаблон, показанный ниже, который состоит из вызовов трех методов (`selectAll()`, `data()` и `join()`) выборки:

```
selection
  .selectAll("selector")
  .data(myData)
  .join("element to add");
```

Мы объясним шаблон привязки данных на примере нашего упражнения с гистограммой. В этой визуализации мы должны показать один прямоугольник для каждой записи в наборе данных. С помощью шаблона привязки данных мы сообщаем D3, что каждый элемент прямоугольника должен соответствовать дате.

Вернемся к `main.js` и внутри функции `createViz()` создадим выборку, включающую контейнер SVG, и сохраним ее в константе с именем `svg`. В нее мы будем добавлять наши прямоугольники. Затем привяжем вызов метода `selectAll()` к выборке и передадим параметр с типом добавляемого элемента, – в данном случае `rect`. В вызов `selectAll()` можно передать любой селектор CSS, но обычно используются типы элементов:

```
const createViz = (data) => {

  svg
    .selectAll("rect")

};
```

Возможно, вы удивлены, что мы выбираем элементы, которые еще даже не существуют! Это то, что мы называем *пустым выбором*. Но D3 пока не знает, сколько прямоугольников нужно добавить, поэтому далее мы включаем в цепочку вызов метода `data()` и передаем ему набор данных. Теперь D3 знает, что должна создать один элемент `rect` для каждой записи в данных:

```
svg
  .selectAll("rect")
  .data(data)
```

В заключение добавляем вызов `join()`, который вставит прямоугольники в DOM:

```
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
```

Сохраните файл и откройте дерево на DOM в инспекторе. Теперь контейнер SVG содержит 33 прямоугольника (рис. 3.16), по одному для каждой технологии в наборе данных.

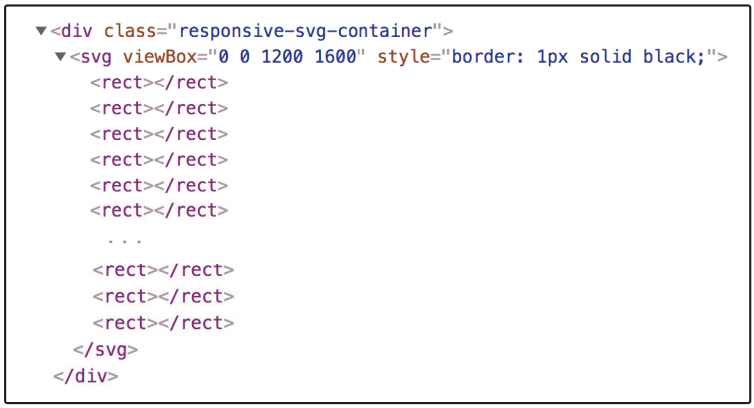


Рис. 3.16 Прямоугольники, привязанные к данным, добавлены в DOM

Рисунок 3.17 иллюстрирует процесс привязки данных. Он начинается с создания выборки, которая в данном примере включает контейнер SVG. Затем вызовом метода `selectAll()` с селектором создается пустая выборка. Далее вызывается метод `data()`, которому передается набор данных. И наконец, вызов `join()` добавляет в дерево DOM по одному прямоугольнику для каждой записи. По завершении привязки данных выборка превращается в *комбинацию* элементов и данных. Всякий раз, обращаясь к элементам из этой выборки, мы получаем доступ к соответствующим им данным!

Еще один шаблон привязки данных

Поискав примеры кода D3 в интернете, вы почти наверняка встретите немного иную модель привязки данных, в которой вместо `.join()` используются методы `.enter().append()`:

```
selection
  .selectAll("selector")
  .data(myData)
  .enter().append("element type");
```

Подход с применением `.enter().append()` по-прежнему актуален, но начиная с D3 версии 6 ему на смену пришел `.join()`.

Метод `join()` не только обрабатывает элементы, добавляемые в выборку на основе данных, но также учитывает, сколько новых элементов вводится в DOM, сколько удаляется и сколько обновляется. Этот более сложный шаблон особенно эффективен в интерактивных визуализациях, где отображаемые данные могут развиваться с течением времени. Заботясь обо всех этих аспектах привязки данных, `join()` проще в использовании, чем предыдущий подход `.enter().append()`.

Этот более сложный подход мы рассмотрим в главе 7. А пока просто помните, что в предыдущих версиях D3 использовался немного другой шаблон привязки данных, и вы, скорее всего, встретите код, который его использует.

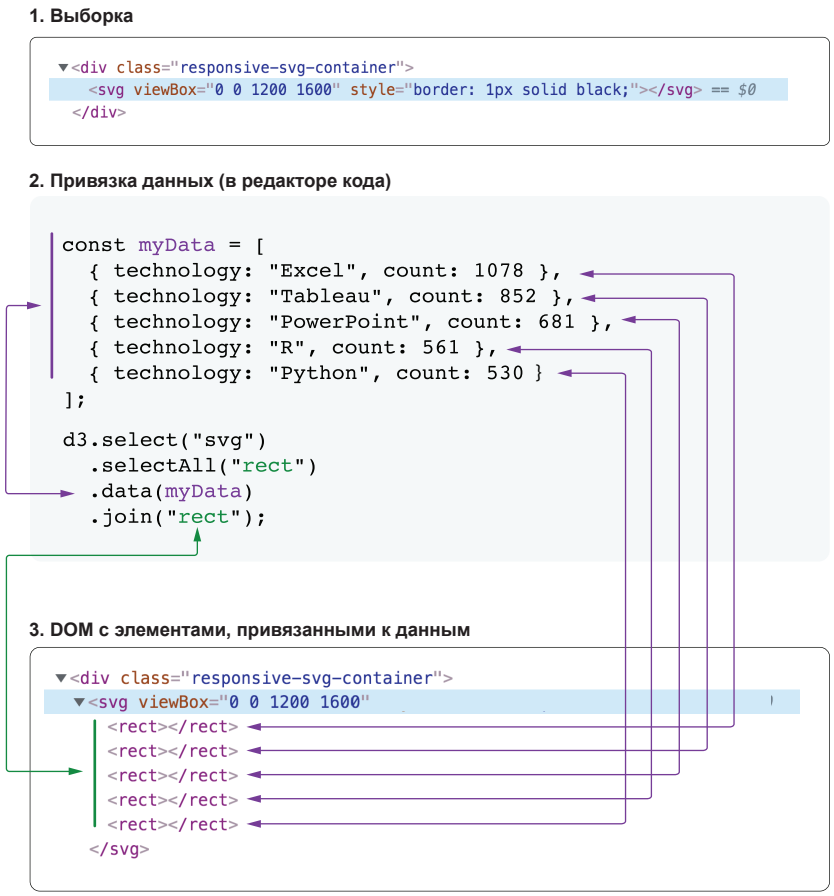


Рис. 3.17 Процесс привязки данных

3.3.1 Динамическая настройка атрибутов DOM с помощью данных

Выше упоминалось, что в процессе загрузки CSV-файла в проект библиотека D3 преобразует его в итерируемую структуру данных – массив объектов. Получив этот массив, мы привязали каждый объект, содержащийся в нем, к элементу `rect`. Привязка данных не только добавила нужное количество прямоугольников в DOM, но и сделала их доступными для манипулирования с применением встроенных функций.

Давайте посмотрим, как осуществляется такое манипулирование на примере нашей столбчатой диаграммы. В конец шаблона привязки данных добавьте вызов метода `attr()`, который будет применен к выборке, содержащей прямоугольники. Используем его для добавления атрибута `class` в каждый прямоугольник, но вместо простого значения передадим во втором параметре функцию доступа.

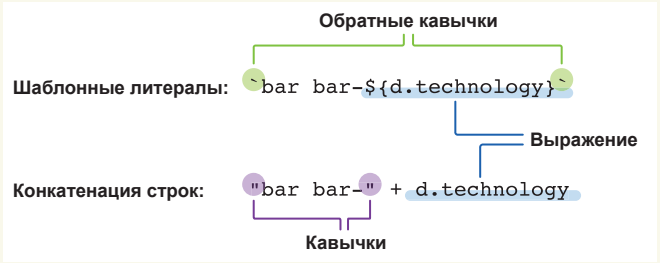
Как можно видеть в следующем фрагменте, функция доступа структурирована так же, как любая функция JavaScript. Здесь она возвращает имя класса `"bar"`, но вы можете вернуть любое другое имя класса для прямоугольников по своему выбору:

```
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    .attr("class", d => {
      console.log(d);
      return `bar bar-${d.technology}`;
    })
```

Функция доступа экспортирует параметр `d`, представляющий запись в данных, привязанную к текущему прямоугольнику. Если вывести `d` в консоль, то вы увидите список объектов, содержащих названия технологий и соответствующие им счетчики, как если бы мы *обошли в цикле* прямоугольники и их данные.

Шаблонные литералы и конкатенация строк

В предыдущем фрагменте мы использовали шаблонные литералы, также известные как шаблонные строки, заключенные в обратные кавычки (``` `` ```). Они используются для объединения традиционных строк JavaScript с выражениями. Выражения предваряются символом доллара и заключаются в фигурные скобки (`${expression}`).



Использование шаблонных литералов вместо конкатенации строк

Возможно, вам более знакома конкатенация строк – более старый, но вполне корректный способ объединения выражений со строками. Как показано на предыдущем рисунке, операция конкатенации объединяет строку, заключенную в кавычки (""), с выражением с помощью знака «плюс» (+). Оба подхода правильные, но применение шаблонных литералов считается более современным.

Этот способ доступа к связанным данным удобен для выбора местоположения и размера каждого прямоугольника. Мы знаем, что хотим расположить прямоугольники вертикально, как показано на рис. 3.18. Атрибут `width` каждого прямоугольника представляет количество специалистов, использующих инструмент, которое хранится в ключе `count` в связанных данных. Чем длиннее прямоугольник, тем популярнее технология, и наоборот. С другой стороны, их атрибут `height` имеет постоянное значение, и между прямоугольниками есть небольшой промежуток.

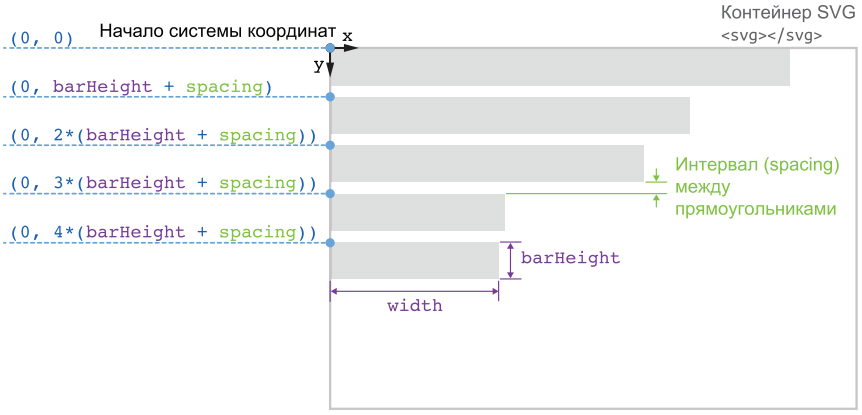


Рис. 3.18 Выбор формулы для определения местоположения верхнего левого угла каждого прямоугольника

Если сохранить высоту столбцов в константе `barHeight`, то мы сможем задать атрибуты `width` и `height` для прямоугольников в выборке, как показано ниже. Обратите внимание, что доступ к значению `count`, привязанному к каждому прямоугольнику, производится с помощью функции обратного вызова при установке атрибута `width`:

```
const barHeight = 20;
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    .attr("class", d => {
      console.log(d);
      return "bar";
    })
    .attr("width", d => d.count)
    .attr("height", barHeight)
```

Далее нам нужно задать позиции прямоугольников, вычислив координаты верхних левых углов и поместив их в атрибуты `x` и `y`. Взглянув на рис. 3.18, можно заметить, что прямоугольники выровнены по левой границе родительского элемента SVG, т. е. атрибут `x` всех прямоугольников равен 0.

Чтобы получить значение для атрибута `y`, нужно выполнить некоторые вычисления. Верхний левый угол первого прямоугольника расположен в верхней части контейнера SVG, т. е. его атрибут `y` имеет значение 0. Второй прямоугольник расположен под первым на расстоянии, равном высоте прямоугольника плюс небольшой интервал (напомню, что ось `y` в системе координат элементов SVG направлена сверху вниз). Третий прямоугольник находится еще ниже, в позиции с координатой `y`, равной высоте двух прямоугольников плюс два вертикальных интервала между ними. На рис. 3.18 мы видим, как формируется закономерность. Позиция `y` каждого прямоугольника соответствует количеству прямоугольников перед ним, умноженному на высоту прямоугольника и вертикальный интервал.

В функции обратного вызова в операции установки атрибута `y` у нас есть доступ ко второму параметру, которому часто дают имя `i` (от англ. *index*). Мы уже говорили, что тело функции доступа подобно телу цикла, перебирающего данные связанных элементов. В циклах JavaScript обычно есть доступ к индексу каждого элемента в массиве минус 1 (нумерация элементов массивов в JavaScript начинается с нуля). В следующем фрагменте мы используем индекс для вычисления вертикального положения каждого прямоугольника и оставляем 5 пикселей пустого пространства между ними:

```
const barHeight = 20;
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    .attr("class", d => {
      console.log(d);
      return "bar";
    })
    .attr("width", d => d.count)
    .attr("height", barHeight)
    .attr("x", 0)
    .attr("y", (d, i) => (barHeight + 5) * i)
```

В роли функций доступа мы используем стрелочные функции JavaScript (синтаксис ECMAScript 6 [ES6]). Когда используется только один параметр, как в примере с атрибутами `class` и `width`, скобки можно опустить. Когда используется несколько параметров, как в случае с атрибутом `y`, их необходимо заключить в скобки, например `(d, i)`. Кроме того, функции доступа, занимающие несколько строк, должны заключаться в фигурные скобки ({}) и завершаться оператором `return` (как в примере с атрибутом `class`), а для простых однострочных функций в этом нет необходимости (как в примере с атрибутом `width`). Эти правила наглядно представлены на рис. 3.19.

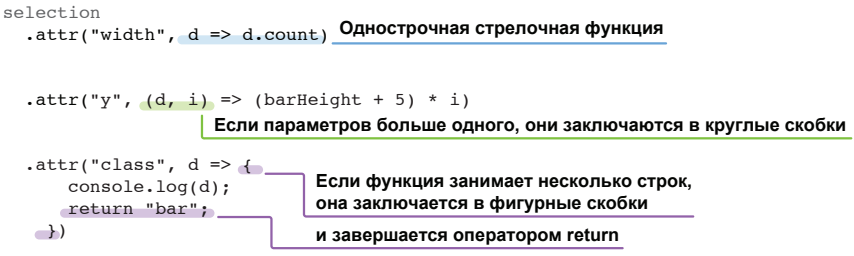


Рис. 3.19 Форматирование стрелочных функций

Сохраните проект, и вы увидите, как прямоугольники займут свое место (рис. 3.20). Результат начинает напоминать столбчатую диаграмму!

> **СОВЕТ** В следующем разделе вы узнаете, что вычислить положение каждого прямоугольника по вертикали можно также с помощью интервальных шкал. Тем не менее умение определить местоположение самостоятельно – ценный навык. При создании проектов D3 нам регулярно приходится делать такие небольшие вычисления координат элементов на экране, поэтому важно уметь решать эту задачу. Поначалу это может быть нелегко, но постепенно вы освоитесь!

Один из лучших способов осваивания таких вычислений – нарисовать несколько элементов из визуализации на листе бумаги и определить их местоположение в системе координат родительского элемента SVG, как это сделано выше на рис. 3.18. Это поможет вам лучше понять, как строятся визуализации, и пригодится при работе над сложными проектами.

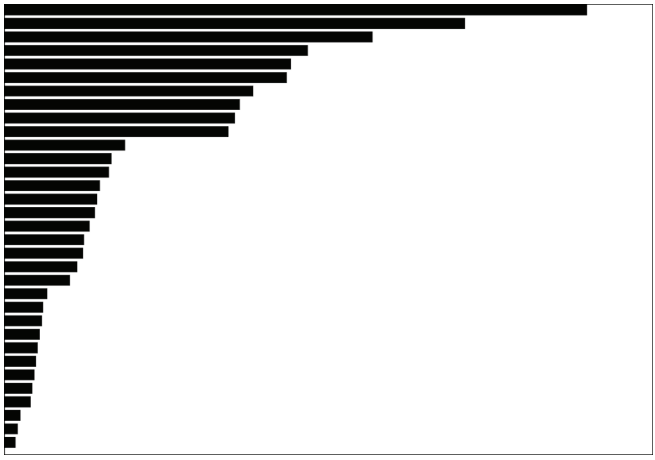


Рис. 3.20 Прямоугольники сформированы и размещены в соответствии с данными

Теперь сделаем нашу диаграмму чуть более привлекательной, залив прямоугольники синим цветом с помощью атрибута `fill`. Следующий фрагмент присваивает этому атрибуту CSS-имя цвета `"skyblue"`, но вы можете использовать другой цвет, если хотите:

```
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    ...
    .attr("fill", "skyblue");
```

В качестве последнего шага опросим данные, привязанные к прямоугольникам, и определим, какой из них соответствует D3.js. Для этого используем тернарный оператор JavaScript и с его помощью сравним название технологии, привязанной к текущему прямоугольнику, со строкой `"D3.js"`. В случае равенства присвоим атрибуту `fill` CSS-цвет `"yellowgreen"`; в противном – цвет `"skyblue"`. Результат показан на рис. 3.21:

```
...
  .attr("fill", d => d.technology === "D3.js" ? "yellowgreen" : "skyblue");
```

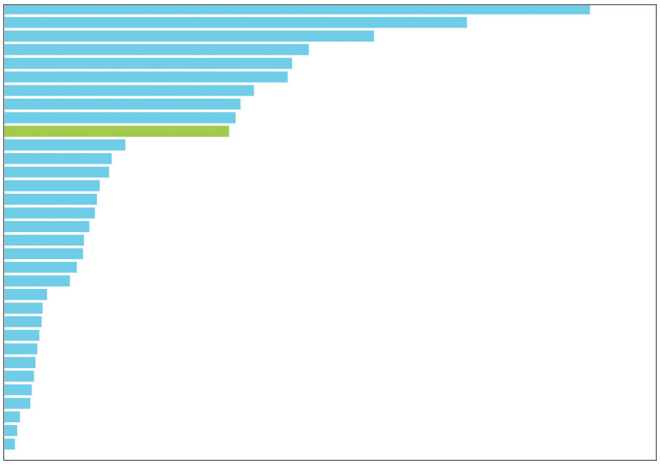


Рис. 3.21 Прямоугольник, соответствующий библиотеке D3.js, окрашен в желто-зеленый цвет, а остальные – в синий

Наша столбчатая диаграмма начинает обретать форму. Сейчас мы напрямую используем данные для определения ширины каждого прямоугольника, но этот подход не всегда практичен. Если бы значения в данных имели большую величину, скажем порядка миллионов, то мы бы не смогли использовать их напрямую. В следующем разделе мы познакомимся со шкалами, которые помогают преобразовывать фактические значения в данных в визуальные атрибуты.

3.4 Адаптация данных под размеры экрана

Создавая визуализации, мы преобразуем данные в визуальные характеристики, такие как размер элемента, цвет или местоположение на экране. В проектах D3 это преобразование выполняется с помощью шкал (последний шаг на рис. 3.22).

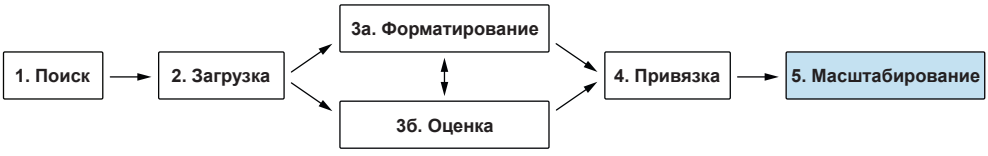


Рис. 3.22 На последнем шаге рабочего процесса D3 используются шкалы для преобразования фактических значений в визуальные характеристики, такие как длина, местоположение и цвет

3.4.1 Шкалы

Шкалы – это функции, принимающие значение из фактических данных и возвращающие результат, который можно напрямую использовать для установки размера, местоположения или цвета элемента визуализации. Более конкретно, входные данные являются частью *предметной области* (domain), определяющей спектр возможных значений в данных. На экране эта предметная область отображается в выходной *диапазон* (range), определяющий спектр возможных выходных значений.

В D3 есть несколько функций шкал, которые принимают интервалы значений предметной области и выходного диапазона. Возьмем для примера функцию линейной шкалы `d3.scaleLinear()`. Чтобы инициализировать функцию шкалы, нужно связать методы `domain()` и `range()`. Метод `domain()` принимает спектр возможных значений фактических данных в форме массива с минимальным и максимальным значениями. А метод `range()` принимает аналогичный массив с соответствующими минимальным и максимальным выходными значениями:

```
const myScale = d3.scaleLinear()
  .domain([0, 200])
  .range([0, 20]);
```

Затем шкалу можно вызывать как любую другую функцию JavaScript. Ей передается значение из предметной области, а она возвращает соответствующее значение из выходного диапазона:

```
myScale(100) => 10
```

Входные и выходные значения шкалы могут быть *непрерывными* или *дискретными*. Непрерывные значения могут существовать в любом месте предопределенного спектра, например число с плавающей точкой от 0 до 100 или дата от июня 2020 года до января 2021 года. Непрерывные количественные данные можно представить как скользящую шкалу значений. Дискретные входные и выходные данные, в свою очередь, могут иметь только строго предопределенный набор значений, например набор футболок с размерами XS, S, M, L и XL, или набор цветов, включающий синий, зеленый, желтый и красный. Работа с качественными данными похожа на укладывание предметов в разные коробки. Каждый предмет может поместиться только в одну коробку.

В D3 количественные данные обычно связываются со шкалой с непрерывной предметной областью. Качественные данные, на-

оборот, подразумевают дискретность предметной области, обычно представляемой в виде массива возможных значений. Следовательно, шкала с непрерывным диапазоном выходных значений может вернуть любое значение в указанных пределах, тогда как шкала с дискретным диапазоном будет возвращать значения только из предопределенного списка.

Основываясь на этих понятиях непрерывных и дискретных входных и выходных данных, шкалы D3 можно сгруппировать в четыре семейства:

- шкалы с непрерывными входными и выходными данными;
- шкалы с дискретными входными и выходными данными;
- шкалы с непрерывными входными и дискретными выходными данными;
- шкалы с дискретными входными и непрерывными выходными данными.

Допустим, мы решили отыскать в Google сериалы, выпущенные в 2021 году, и сгруппировать первые 10 результатов в набор данных (мы выполняли это упражнение, когда писали эту главу). Затем мы должны извлечь информацию о жанре каждого сериала, его рейтинге на Rotten Tomatoes, средней оценке профессиональных критиков и платформе, где он доступен. Этот набор данных, показанный в табл. 3.2, содержит как непрерывные значения (оценки зрителей и критиков), так и дискретные (жанры и платформы). Чтобы создать столбчатую диаграмму, изображенную на рис. 3.23, нам нужно использовать шкалу D3 из всех семейств, перечисленных выше. Обратите внимание, что вам не нужно строить эту столбчатую диаграмму; ее цель – просто проиллюстрировать различные типы шкал и подсказать, в каких случаях их использовать.

Таблица 3.2 Результаты поиска в Google телесериалов, выпущенных в 2021 году

Название	Жанр	Оценка зрителей (%)	Оценка критиков (%)	Платформа
Девять совсем незнакомых людей	Драма	59	62	Prime
Уборщица. История матери-одиночки	Драма	88	94	Netflix
Катла	Драма	78	100	Netflix
Наследие Юпитера	Боевик	73	40	Netflix
Неслучайность	Боевик	72	82	Netflix
Нерегулярные части	Детектив	54	80	Netflix
Тень и кость	Боевик	89	88	Netflix
Кликбейт	Детектив	64	56	Netflix
Секс/жизнь	Комедия	34	23	Netflix
Колесо времени	Боевик	64	82	Prime

Источник: www.rottentomatoes.com

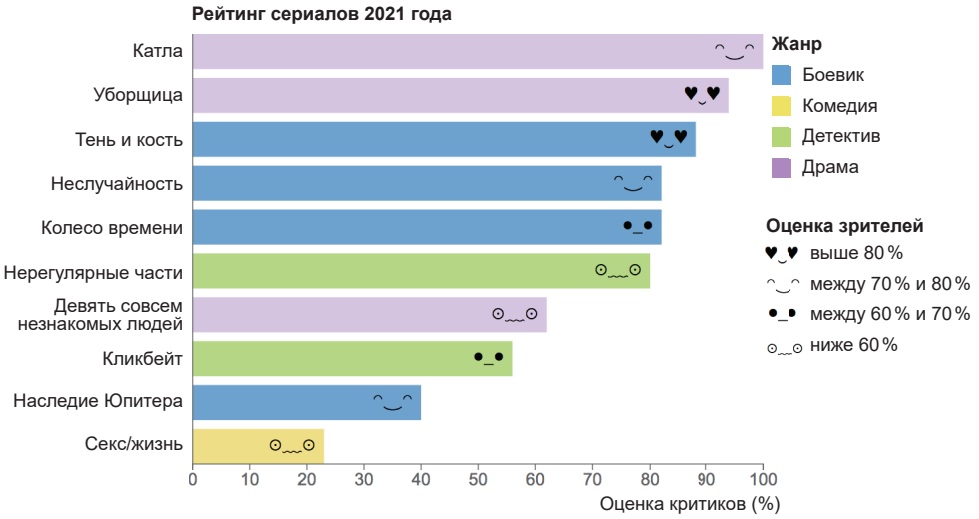


Рис. 3.23 На последнем этапе рабочего процесса D3 с помощью шкал производится преобразование значений данных в визуальные атрибуты, такие как длина, местоположение и цвет

На столбчатой диаграмме каждый прямоугольник соответствует сериалу, а его длина пропорциональна средней оценке, данной критиками. Для расчета длины прямоугольников мы используем шкалу D3 из первого семейства, которая принимает непрерывное значение (оценку критиков) и возвращает непрерывное значение (длину соответствующего прямоугольника).

Предметная область шкалы включает значения от 0 % до 100 % (спектр возможных оценок критиков), а диапазон определяет длину столбцов в пикселях:

```
domain => [0, 100] возможные минимальное и максимальное значения на входе в %
range  => [0, 500] соответствующие минимальное и максимальное значения на выходе
          в пикселях
```

Цвет прямоугольников представляет жанр каждого сериала. Чтобы придать прямоугольникам соответствующий цвет, нужна шкала из второго семейства, которая принимает дискретное значение на входе (жанр) и возвращает дискретное значение на выходе (соответствующий цвет). Предметная область этой шкалы – массив жанров, а диапазон – массив соответствующих цветов:

```
domain => ["Драма", "Боевик", "Детектив", "Комедия"] возможные входные данные
range  => ["purple", "blue", "green", "yellow"] соответствующие выходные данные
```

Смайлик на конце каждого прямоугольника представляет среднюю оценку зрителей. Сериалы, получившие оценку выше 80 %,

отмечаются смайликом с сердечками, получившие оценку от 70 % до 80 % отмечаются улыбающимся смайликом, получившие оценку от 60 % до 70 % отмечаются нейтральным смайликом, а сериалы с оценкой ниже 60 % получают недовольный смайлик. В данном случае мы имеем непрерывную предметную область (оценка зрителей) и дискретный набор смайликов:

```
domain => [60, 70, 80] пороги для входных значений
range  => ["o_ _o", "•_•", "-‿-", "♥‿♥"] соответствующие выходные данные
```

Наконец, прямоугольники располагаются вдоль вертикальной оси графика. Для расчета местоположения каждого прямоугольника нужна шкала из четвертого семейства, которая принимает дискретные значения (названия сериалов) и возвращает непрерывные выходные данные (местоположение вдоль вертикальной оси):

```
domain => ["Девять совсем незнакомых людей", "Уборщица", "Катла", ...] список
          входных значений
range  => [0, 500] соответствующие минимальное и максимальное значения на выходе
          в пикселях
```

Каждое семейство шкал содержит несколько функций. На момент написания этой книги в модуле d3-scale (https://d3js.org/d3-scale) имелось более 20 таких функций. В этой главе мы познакомимся с функциями `d3.scaleLinear()` и `d3.scaleBand()` как наиболее часто используемыми в проектах D3 и необходимыми для завершения реализации диаграммы. Но далее в книге мы рассмотрим множество других шкал. В приложении B вы найдете обзор всех доступных функций шкал и дерево решений, которое поможет вам выбрать правильную шкалу.

3.4.2 Линейная шкала

Чаще всего в проектах D3 используется линейная шкала (`d3.scaleLinear()`). Она принимает непрерывную предметную область на входе данных и возвращает непрерывный диапазон на выходе:

```
const myLinearScale = d3.scaleLinear()
  .domain([0, 250])
  .range([0, 25]);
```

Выходное значение линейной шкалы прямо пропорционально входному, как показано на рис. 3.24. В предыдущем фрагменте предметная область охватывает значения от 0 до 250, а диапазон выходных данных – значения от 0 до 25. Если вызвать эту функцию с аргументом 100, то она вернет 10. Аналогично если передать ей значение 150, она вернет 15:

```
myLinearScale(100) => 10
myLinearScale(150) => 15
```

Вернемся к нашему упражнению с гистограммой. В предыдущем разделе мы использовали значения счетчиков из данных для установки атрибута ширины каждого прямоугольника. У нас это получилось, потому что счетчики были относительно небольшими числами, но, как правило, для преобразования значений из данных в атрибуты SVG практичнее использовать шкалы.

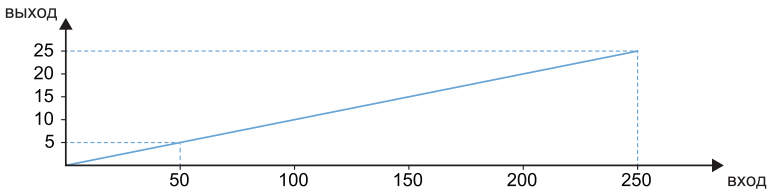


Рис. 3.24 Выход линейной шкалы линейно пропорционален входу

Для иллюстрации предположим, что контейнер SVG визуализации имеет размер 600×700 пикселей, а не 1200×1600 пикселей. Измените настройку атрибута `viewbox` контейнера SVG в `main.js`, чтобы отразить это новое соотношение ширины и высоты:

```
const svg = d3.select(".responsive-svg-container")
  .append("svg")
  .attr("viewBox", "0 0 600 700")
  ...
```

Также измените значение свойства `max-width` элемента `div` с классом `responsive-svg-container` в `main.css`:

```
.responsive-svg-container {
  ...
  max-width: 600px;
  ...
}
```

Сохраните проект, перейдите в браузер – и вы увидите, что первые три прямоугольника на диаграмме вышли за границы контейнера SVG и их концы оказались скрыты. Мы исправим эту проблему с помощью линейной шкалы, которая отобразит значения счетчиков в доступное пространство в контейнере SVG, оставив при этом свободное место для меток.

Сначала объявим константу с именем `xScale`. Эта шкала будет отвечать за размер и позиционирование элементов по оси `x`. Затем вызовем функцию `d3.scaleLinear()` и добавим в цепочку методы `domain()` и `range()`.

Диапазон возможных значений `count` в нашем наборе данных простирается от 0, теоретического минимума, до 1078, самого большого значения `count`, соответствующего технологии Excel. Обратите внимание, что вместо фактического минимального значения `count` в наборе данных мы используем 0. Как и в большинстве диаграмм, нам нужно, чтобы ось `x` начиналась с 0. Передадим минимальное и максимальное значения в предметной области в вызов метода `domain()` в форме массива (`[0, 1078]`).

Теперь оценим доступное пространство по горизонтали – диапазон шкалы. На рис. 3.25 показаны первые пять прямоугольников диаграммы. У вас не должны быть видны метки слева и справа на диаграмме. Мы добавили их на эту иллюстрацию, только чтобы показать, зачем нужно дополнительное пространство.

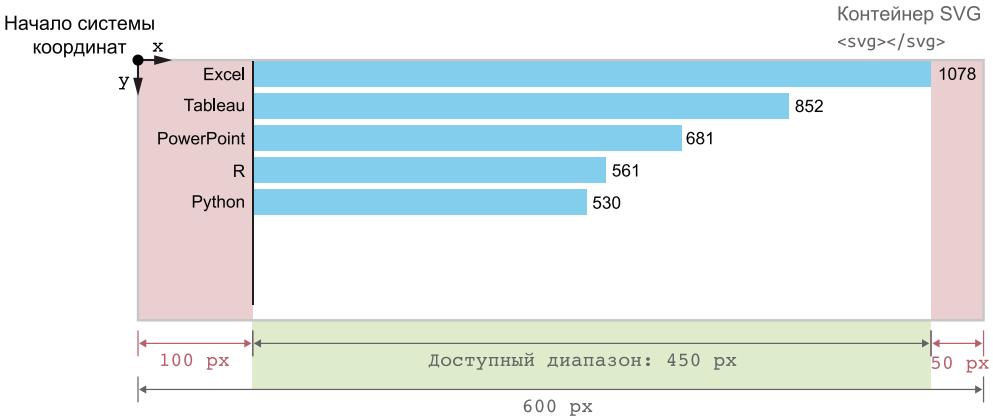


Рис. 3.25 Оценка пространства по горизонтали, доступного для прямоугольников

Мы уже знаем, что контейнер SVG имеет ширину 600 пикселей. Нам нужно оставить 100 пикселей слева для меток с названиями технологий и 50 пикселей справа для меток с числами. Это означает, что прямоугольники могут иметь ширину от 0 до 450 пикселей.

Теперь объявим `xScale` с предметной областью, изменяющейся от 0 до 1078, и диапазоном от 0 до 450. Добавим линейную шкалу внутри функции `createViz()` перед привязкой данных:

```
const createViz = (data) => {

  const xScale = d3.scaleLinear()
    .domain([0, 1078])
    .range([0, 450]);

  // Привязка данных
  ...
}
```

Выше уже говорилось, что шкалы D3 можно вызывать как обычные функции JavaScript. Им передается значение из предметной области, а они возвращают соответствующее значение из заданного диапазона. Например, если передать в `xScale` значение `1078`, соответствующее значению счетчика Excel, то шкала вернет `450`. Если передать `414`, количество практиков, использующих D3, то шкала вернет `172.82`, ширину в пикселях прямоугольника, соответствующего D3.js:

```
xScale(1078)   // => 450
xScale(414)    // => 172.82
```

Попробуйте сами: войдите в консоль, получите результаты, возвращаемые шкалой, для нескольких значений из набора данных и проверьте, соответствуют ли они значениям на рис. 3.26.

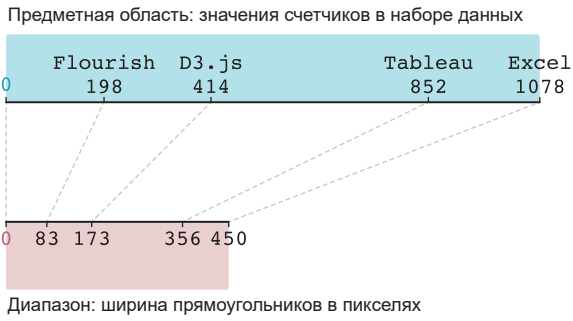


Рис. 3.26 Отображение линейной шкалой значений счетчиков из данных в ширину столбцов

Теперь, объявив шкалу, ее можно использовать для вычисления ширины каждого прямоугольника в нашей диаграмме. Найдите строку кода, где задается атрибут `width` прямоугольников, и, как показано в следующем фрагменте, вместо использования значения `count` напрямую вызовите `xScale()` и передайте значение `count` в качестве аргумента. Измените значение атрибута `x` на 100, чтобы сместить прямоугольники вправо, дабы оставить место для меток с названиями технологий, как показано на рис. 3.25:

```
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    ...
    .attr("width", d => xScale(d.count))
    ...
    .attr("x", 100)
    ...
```

Сохраните проект и посмотрите, что теперь прямоугольники вписываются в контейнер SVG и сохраняется пустое пространство для надписей по обе стороны от них.

Теперь вы знаете, как использовать шкалы D3! Несмотря на большое разнообразие шкал в D3, все они объявляются и используются схожим образом. Для переключения с одной шкалы на другую нужно знать только тип данных предметной области и ожидаемый диапазон выходных значений.

3.4.3 Интервальная шкала

Второй тип шкал, который нам понадобится для нашей диаграммы, – интервальная шкала. Интервальные шкалы относятся к четвертому семейству: они принимают дискретные и возвращают непрерывные данные. Интервальную шкалу удобно использовать для равномерного распределения прямоугольников столбчатой диаграммы в пределах доступного пространства.

Чтобы объявить интервальную шкалу, нужно вызвать функцию `d3.scaleBand()`. Следующий фрагмент кода сохраняет шкалу в константу с именем `yScale`. Такое имя выбрано потому, что эта шкала отвечает за распределение элементов по оси `y`. Предметная область этой шкалы – массив, содержащий все технологии из нашего набора данных. Этот массив легко сгенерировать с помощью JavaScript-функции `map()`. (Вернитесь к разделу 1.2.5, чтобы вспомнить, когда и как используется функция `map()`.) Диапазон охватывает все доступное пространство по вертикали: от 0 пикселей в верхней части контейнера SVG до 700 пикселей в нижней:

```
const yScale = d3.scaleBand()
  .domain(data.map(d => d.technology))
  .range([0, 700]);
```

Добавьте интервальную шкалу внутрь `createViz()` перед привязкой данных. При вызове с названием технологии из набора данных эта шкала возвращает местоположение соответствующего прямоугольника по вертикали. Например, если передать строку `"Excel"` в `yScale`, то она вернет `0`, потому что прямоугольник, соответствующий Excel, находится в верхней части диаграммы. Аналогично если вызвать `yScale`, передав ей строку `"D3.js"`, то она вернет `272.72` – координату `y` верхнего левого угла прямоугольника, соответствующего D3:

```
yScale("Excel")   // => 0
yScale("D3.js")   // => 272.72
```

Помните, сколько вычислений нам приходилось выполнять ранее, чтобы задать атрибуты `y` прямоугольников? Благодаря интервальной шкале мы теперь можем с легкостью задать этот атрибут, передав в `yScale` имя технологии, привязанной к каждому прямоугольнику:

```
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    ...
    .attr("y", d => yScale(d.technology))
    ...
```

Интервальные шкалы также имеют очень удобный метод `bandwidth()`, возвращающий высоту прямоугольников, которая пропорциональна количеству прямоугольников и доступному пространству. В нашей диаграмме эта высота соответствует атрибуту `height` прямоугольников. В следующем фрагменте показано, как вызовом метода `bandwidth()` интервальной шкалы можно получить значение для атрибута `height`:

```
svg
  .selectAll("rect")
  .data(data)
  .join("rect")
    ...
    .attr("height", yScale.bandwidth())
    ...
```

Сохраните проект и откройте его в браузере. Как показано на рис. 3.27, прямоугольники покрывают все вертикальное пространство, доступное в контейнере SVG, но отсутствие промежутков затрудняет восприятие графика.

Эту проблему легко исправить, установив свойство `paddingInner()` интервальной шкалы, которое определяет величину отступа между прямоугольниками полосой и принимает значения от 0 до 1. Здесь мы задаем ему значение `0.2`, что составляет 20 % от высоты прямоугольников:

```
const yScale = d3.scaleBand()
  .domain(data.map(d => d.technology))
  .range([0, 700])
  .paddingInner(0.2);
```

После этого макет нашей столбчатой диаграммы станет чуть более простым для восприятия, как можно видеть на рис. 3.28. Так намного лучше!

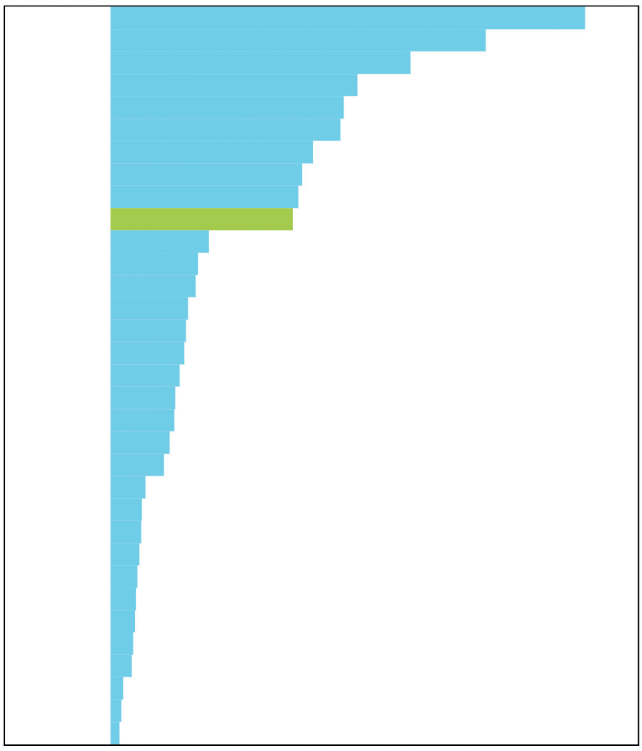


Рис. 3.27 Распределение прямоугольников с применением интервальной шкалы без промежутков

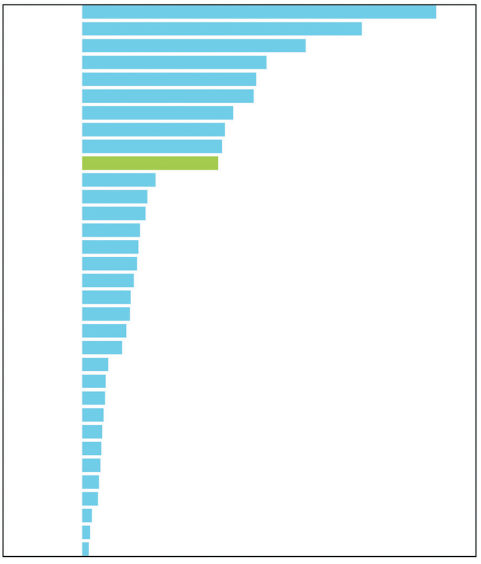


Рис. 3.28 Распределение прямоугольников с применением интервальной шкалы с промежутками

Схема на рис. 3.29 кратко описывает работу интервальной шкалы. Она принимает предметную область в виде списка технологий из нашего набора данных и равномерно распределяет их в вертикальном пространстве контейнера SVG. Координату y верхнего левого угла каждого прямоугольника можно получить, вызвав функцию шкалы и передав ей название технологии (`yScale("PowerPoint")`). Аналогично можно получить высоту прямоугольников, вызвав метод `bandwidth()` шкалы (`yScale.bandwidth()`). Наконец, по умолчанию промежуток между столбцами равен 0. Мы можем сообщить D3 желаемую величину промежутка, присвоив свойству `paddingInner()` интервальной шкалы значение от `0` до `1`.

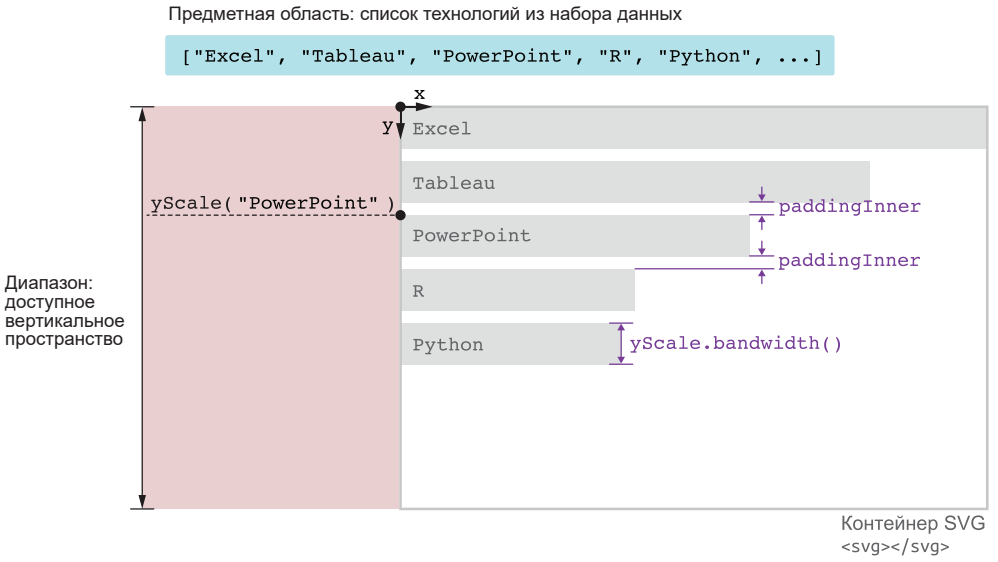


Рис. 3.29 Как интервальная шкала распределяет список технологий в пределах доступного вертикального пространства

3.5 Добавление меток на диаграмму

Мы почти закончили создание столбчатой диаграммы, но прямо сейчас невозможно узнать, какой прямоугольник какой технологии соответствует и какие значения выражаются шириной прямоугольников. Мы исправим этот недостаток, добавив два набора меток. Первый набор будет содержать названия технологий, перечисленных с левой стороны. Второй – значения счетчиков, связанных с каждым столбцом и расположенных на концах прямоугольников.

В визуализациях на основе SVG метки создаются с помощью текстовых элементов. Мы объединим с каждым прямоугольником два

текстовых элемента и вложим каждый прямоугольник с его метками в группу SVG. Как рассказывалось в разделе 1.2.2, группы позволяют перемещать несколько элементов как одно целое. Их также можно использовать для передачи связанных данных потомкам, как будет показано далее.

Начнем с небольшой реорганизации кода. Сначала закомментируем все строки, связанные с атрибутами прямоугольников. Мы вернемся к ним чуть ниже. В JavaScript однострочные комментарии начинаются с двух символов слеша (//), а многострочные начинаются парой символов /* и заканчиваются парой символов */.

Теперь вернемся к привязке данных. Вместо привязки данных к прямоугольникам используем группы SVG (g). Кроме того, сохраним выборку в константе с именем barAndLabel:

```
const barAndLabel = svg
  .selectAll("g")
  .data(data)
  .join("g");
```

Чтобы прямоугольники и их метки двигались вместе, применим вертикальное перемещение к каждой группе с помощью атрибута transform. Свойство translate атрибута transform принимает два параметра: перемещение по горизонтали, в котором мы передаем 0, и перемещение по вертикали, соответствующее местоположению прямоугольника по вертикали. Обратите внимание, что для определения этого местоположения вызывается функция yScale, как мы делали это ранее для отдельных прямоугольников:

```
const barAndLabel = svg
  .selectAll("g")
  .data(data)
  .join("g")
    .attr("transform", d => `translate(0, ${yScale(d.technology)})`);
```

Группы SVG не имеют визуального представления и никак не ограничивают пространство, тем не менее их можно представить как блоки, которые вмещают все свои дочерние элементы. Благодаря атрибуту transform группы распределяются в контейнере SVG по вертикали, как показано на рис. 3.30. Координаты прямоугольников и меток определяются относительно координат родительской группы.

Теперь, создав группы, добавим прямоугольники обратно. Вызовем выборку barAndLabel и добавим в нее элементы прямоугольников:

```
const barAndLabel = svg
  .selectAll("g")
```

```
  .data(data)
  .join("g")
    .attr("transform", d => `translate(0, ${yScale(d.technology)})`);

barAndLabel
  .append("rect");
```

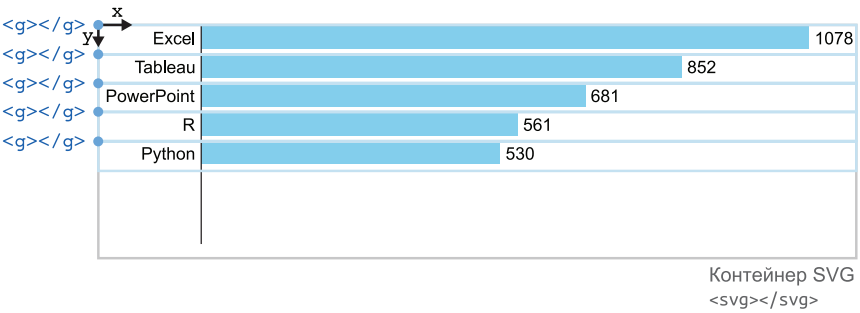


Рис. 3.30 Позиционирование групп и вложенных в них прямоугольников и меток внутри контейнера SVG

Поскольку выборка содержит несколько групп элементов, по одной для каждой точки данных, D3 понимает, что должна добавить один элемент прямоугольника в каждую группу. Сохраните проект и посмотрите на разметку с помощью инспектора. Убедитесь, что группы и прямоугольники были добавлены в DOM, как показано на рис. 3.31.

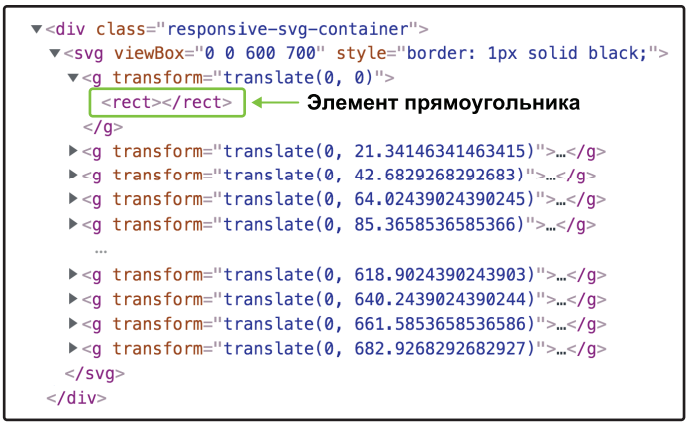


Рис. 3.31 Элементы прямоугольников, объединенные в группы

Теперь можно раскомментировать методы атрибутов прямоугольников и применить их ко вновь добавленным элементам `rect`. Одна из замечательных особенностей привязки данных – они пере-

даются элементам-потомкам групп. У нас по-прежнему есть доступ к данным, как и раньше. Единственное отличие в том, что поскольку вертикальное перемещение уже применено к группам, атрибут y прямоугольников можно установить равным 0:

```
barAndLabel
  .append("rect")
    .attr("width", d => xScale(d.count))
    .attr("height", yScale.bandwidth())
    .attr("x", 100)
    .attr("y", 0)
    .attr("fill", d => d.technology === "D3.js" ? "yellowgreen":"skyblue");
```

Прямоугольники больше не нуждаются в позиционировании по вертикали. Их положение определяется относительно положения родительской группы

Теперь прямоугольники должны быть видны на гистограмме и выглядеть точно так же, как и раньше (рис. 3.28).

Сейчас можно добавить метки! Снова вызовем выборку `barAndLabel` и добавим в нее текстовый элемент. Эта операция добавит текстовый элемент в каждую группу. Нам нужно отобразить название технологии, связанной с каждым прямоугольником. Для этого добавим в цепочку вызов метода `text()`. Он принимает один параметр: текст для добавления в текстовый элемент SVG. Здесь мы выбираем текст динамически из данных, привязанных к каждому элементу:

```
barAndLabel
  .append("text")
    .text(d => d.technology);
```

Затем позиционируем каждую метку, используя атрибуты `x` и `y` текстовых элементов. По горизонтали конец меток должен совпадать с началом прямоугольников. Поскольку прямоугольники начинаются на 100 пикселей правее оси `x`, метки должны заканчиваться примерно на 96 пикселях, чтобы между концом метки и началом соответствующего прямоугольника оставался небольшой промежуток. Мы также используем атрибут `text-anchor` со значением `end`, чтобы выровнять метки по правому краю. Это означает, что атрибут `x` представляет положение конца каждой метки (рис. 3.32).

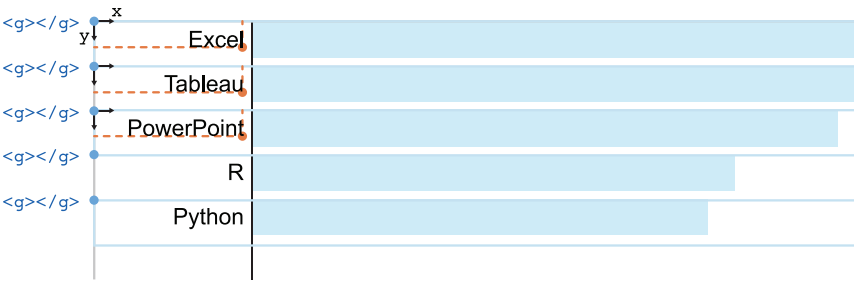


Рис. 3.32 Расчет положения меток с названиями технологий

По вертикали каждая метка позиционируется относительно родительской группы. Их нужно сместить немного вниз, чтобы отцентрировать с прямоугольниками, не забывая при этом, что вертикальное позиционирование текстовых элементов производится по их базовой линии. Здесь применяется сдвиг на 12 пикселей. Обратите внимание, что значения атрибутов `x` и `y` берутся не из воздуха. Мы вычислили их, имея приблизительное представление о том, где должны отображаться метки, и опробовали несколько значений, пока не нашли правильные. Инспектор в браузере – отличное место для проведения таких быстрых экспериментов:

```
barAndLabel
  .append("text")
    .text(d => d.technology)
    .attr("x", 96)
    .attr("y", 12)
    .attr("text-anchor", "end");
```

Наконец, с помощью метода `style()` можно задать свойства `font-family` и `font-size` меток, исходя из своих предпочтений. В этом примере использован шрифт sans-serif размером 11 пикселей. Результат можно увидеть на рис. 3.33:

```
barAndLabel
  .append("text")
    .text(d => d.technology)
    .attr("x", 96)
    .attr("y", 12)
    .attr("text-anchor", "end")
    .style("font-family", "sans-serif")
    .style("font-size", "11px");
```

Теперь мы можем добавить метки на концах прямоугольников, показывающие, сколько раз каждая технология была выбрана в опросе. Процедура очень похожа на ту, что используется для вывода меток с названиями технологий. Сначала нужно вызвать константу `barAndLabel`, содержащую выбранные группы, и с помощью метода `text()` добавить в каждую группу еще один текстовый элемент. Текстом для каждой метки служит строковое представление соответствующего значения `count`:

```
barAndLabel
  .append("text")
    .text(d => d.count)
```

Поскольку метки с количествами располагаются на конце каждого прямоугольника, их горизонтальную координату можно вычислить с помощью той же шкалы `xScale()`, которая возвращает ширину

прямоугольников, добавив небольшой отступ (4 пикселя) и учтя отступ в 100 пикселей слева от прямоугольников. По вертикали метки с количествами тоже сдвинуты вниз на 12 пикселей, как показано на рис. 3.34:

```
barAndLabel
  .append("text")
    .text(d => d.count)
    .attr("x", d => 100 + xScale(d.count) + 4)
    .attr("y", 12)
```

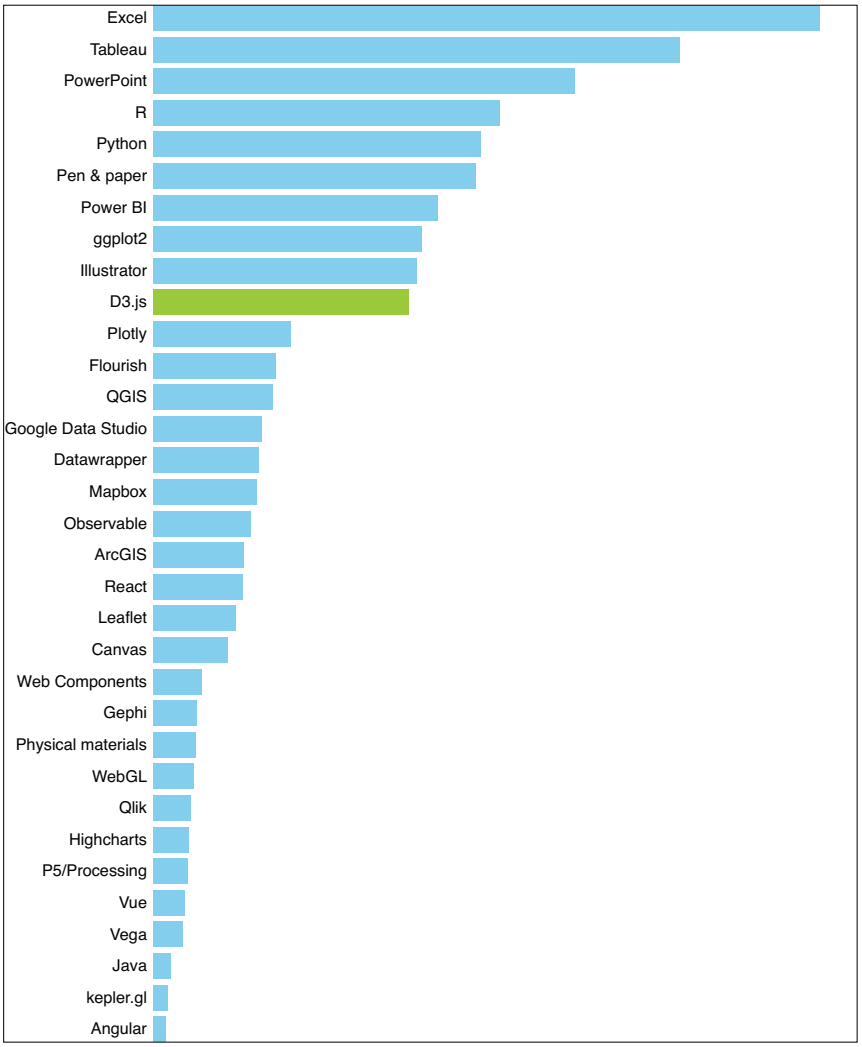


Рис. 3.33 Метки с названиями технологий на столбчатой диаграмме

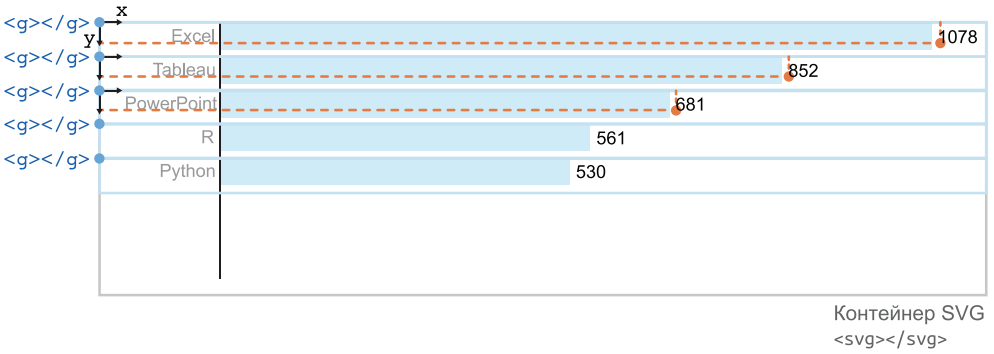


Рис. 3.34 **Расчет местоположений меток с количествами**

Наконец, с помощью метода `style()` можно задать свойства `font-family` и `font-size` меток. Обратите внимание, что размер шрифта меток с количествами (9 пикселей) меньше размера шрифта меток с названиями технологий (11 пикселей). Это сделано, чтобы сохранить визуальную иерархию между двумя типами меток. Более крупные метки привлекут внимание в первую очередь, и зрители поймут, что метки с количествами вторичны по отношению к меткам технологий:

```
barAndLabel
  .append("text")
    .text(d => d.count)
    .attr("x", d => 100 + xScale(d.count) + 4)
    .attr("y", 12)
    .style("font-family", "sans-serif")
    .style("font-size", "9px");
```

В качестве последнего шага добавим вертикальную линию слева от прямоугольников, которая будет изображать вертикальную ось. В следующем фрагменте мы добавили линию в контейнер SVG. Начальная позиция линии (`x1, y1`) находится в точке (`100, 0`), в верхней части контейнера SVG, а конечная позиция (`x2, y2`) – в точке (`100, 700`), в нижней части контейнера. Также нужно указать цвет рамки, чтобы сделать линию видимой:

```
svg
  .append("line")
    .attr("x1", 100)
    .attr("y1", 0)
    .attr("x2", 100)
    .attr("y2", 700)
    .attr("stroke", "black");
```

Если удалить границу из контейнера SVG, то столбчатая диаграмма должна выглядеть, как показано на рис. 3.35 и в проекте на GitHub (http://mng.bz/mjor). Обратите внимание, что подход к оформлению промежутков между метками и прямоугольниками, использованный в этой главе, не является общепринятым. Обычно мы используем соглашение о границах, принятое в D3. Оно будет представлено в следующей главе и использоваться в остальной части книги.

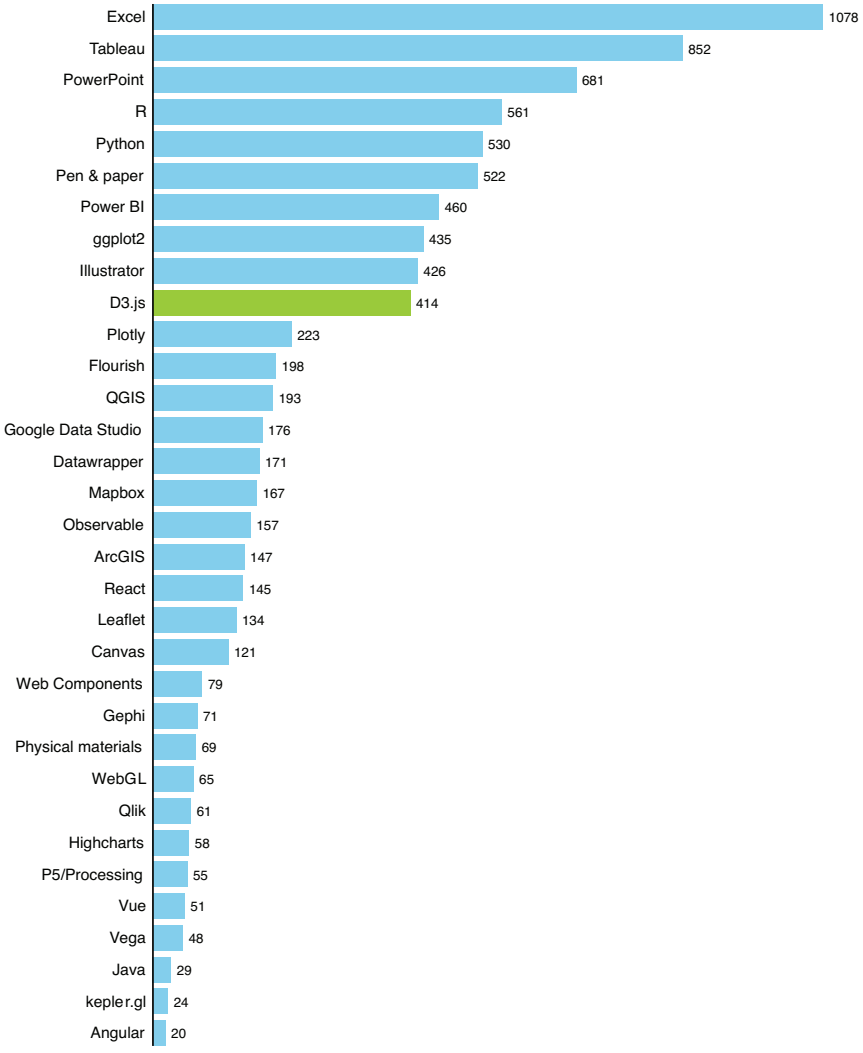


Рис. 3.35 Окончательная столбчатая диаграмма (http://mng.bz/mjor)

Вы дочитали эту очень насыщенную главу до конца, с чем вас и поздравляем! Не волнуйтесь, если какие-то из обсуждавшихся концепций остались вами непонятыми. Мы продолжим демонстри-

ровать эти концепции в разных контекстах, и вскоре они станут вашей второй натурой.

Интервью с Кристиной Сюч

Сюч – дизайнер в сфере визуализации данных.

Расскажите немного о себе, как вы открыли для себя D3.js?

У меня нет опыта программирования, но есть степень магистра в графическом дизайне, и мне довелось немного использовать ActionScript в эпоху Flash. Сначала я использовала dataviz, но в 2012 году узнала, что многие применяют эту новую штуку под названием d3.js и что она очень хорошо подходит для визуализации данных, поэтому я решила попробовать и изучить ее.

Как вы изучали D3? С какими основными трудностями столкнулись и как их преодолели?

Я не любительница изучать новые библиотеки или языки. Предпочитаю выучить что-то одно, а потом использовать это вечно. Обычно я очень расстраиваюсь, когда появляется что-то новое и затем получает широкое распространение, из-за чего я вынуждена тратить время на изучение.

Когда появилась библиотека d3.js, все начали говорить о ней. Я уже осознала, что хочу создавать интерактивные визуализации, поэтому решила, что она будет единственной библиотекой, которую я изучу.

Это было сложнее, чем я ожидала. Я читала руководства из *Interactive Data Visualization for the Web* Скотта Мюррея (Scott Murray), а затем пыталась применить новые знания в своих проектах. Все было хорошо, пока я в точности следовала каждому шагу, описанному в руководствах, но стоило мне подставить в примеры свои данные, все рушилось, потому что мои данные по большей части были намного сложнее. Поэтому мне пришлось приложить немало усилий, чтобы заставить их работать с d3.js.

Кроме того, с самого начала я хотела создавать большие и необычные визуальные эффекты, но по этому аспекту не было никаких специальных руководств и мне было трудно понять, как перейти от простых столбчатых диаграмм к нетрадиционным, сложным визуализациям, которые я рисовала себе в голове. Я просто практиковалась, пока, наконец, не получала результат, в котором каждая часть выглядела так, как я себе представляла.

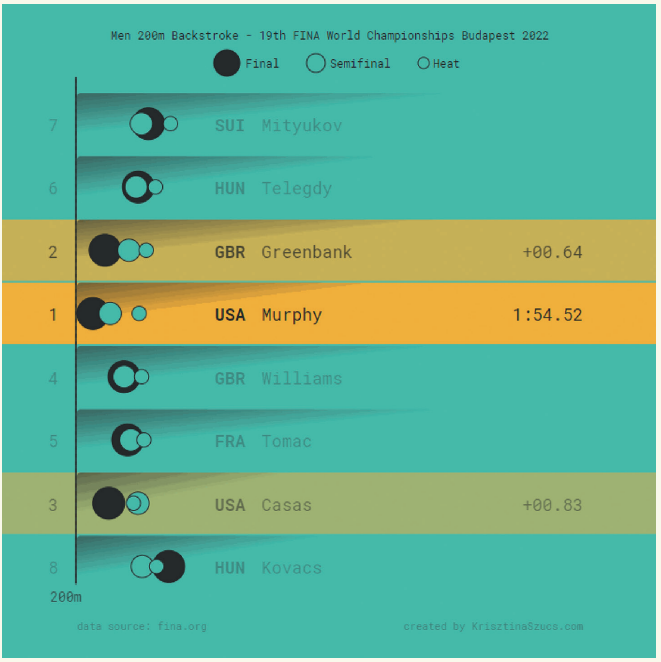
Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в ваших текущих проектах визуализации данных?

Для разработки внешнего вида моих визуализаций я использую такие инструменты, как Figma, а для сбора и обработки данных – Excel и Python.

Используя инструменты для создания дизайна, я переключаюсь на дизайнерское мышление и не думаю об ограничениях, с которыми при-

дется столкнуться, или о проблемах, которые придется решить, когда наступит момент заняться фактической реализацией визуализации. Это помогает мне не ограничивать свои идеи и создавать хорошие визуальные эффекты. Закончив создание визуального образа в форме статичной картинки, я переключаюсь на «мышление разработчика» и просто пытаюсь воссоздать все так, как мне это виделось. Иногда я вношу коррективы, но, проектируя что-то сложное для разработки, я заставляю себя решать проблемы реализации, а не возвращаться и менять дизайн на что-то более простое. Я также обнаружила, что мне это помогает лучше изучить D3.js.

Представленный проект: анимированные спортивные результаты (https://krisztinaszucs.com/)



Анимированная визуализация соревнований по плаванию на спине на 200 м среди мужчин на чемпионате мира по водным видам спорта 2022 года

Опишите ваш проект «Animated Sport Results».

Я планировала создать проект для Олимпийских игр 2020 года. Это крупное мировое событие, которым интересуются многие, а данные о спортивных результатах – доступный ресурс.

Я подумала, что если создам одну визуализацию состязаний, то смогу затем использовать ее для других спортивных мероприятий, таких как чемпионаты мира, или даже для других спортивных дисциплин, использующих схожие правила подсчета очков.

Я также хотела, чтобы зрители вновь пережили волнение борьбы, не раскрывая окончательный результат с самого начала; я хотела показать, как развивалось состязание.

Мне нравятся похожие визуализации с Chartball.com и *New York Times*, и они послужили источником моего вдохновения для этого проекта.

Расскажите, как создавался этот проект?

Когда я начала работу над этим проектом, я знала, что буду создавать анимацию в минималистическом стиле и в оттенках красного цвета, который у меня ассоциируется с Японией (Олимпиада 2020 года проходила в Токио).



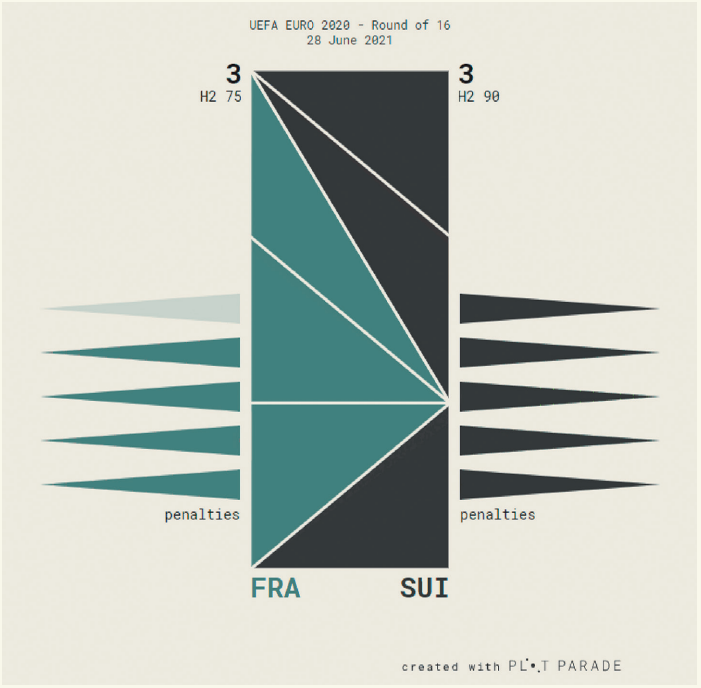
Несколько визуализаций, созданных Кристиной Сюч для проекта «Animated Sport Results»

Но я не знала, какое спортивное событие буду визуализировать, поэтому создала кучу эскизов для разных спортивных дисциплин, таких как прыжки в высоту и в длину, метание копья и молота, бег, велогонки на треке и т. д. …а также кое-что для водного поло, а именно визуализацию накопления очков (нижнее правое изображение на рисунке выше). Я быстро поняла, что визуализацию накопления очков можно использовать чаще всего, и как только я получила удовлетворяющий меня дизайн, я перешла к реализации.

Это случилось за несколько месяцев до Олимпиады, поэтому у меня не было никаких данных и я использовала результаты предыдущих олимпийских игр в качестве образца. Я также создала для себя пользовательский интерфейс, вмещающий основную визуализацию, позволяющий легко и быстро переключаться между цветовыми палитрами или наборами данных.

Визуализация была готова за несколько недель до Олимпиады. Однако примерно в то же время прошел чемпионат Европы по футболу УЕФА,

поэтому я попыталась протестировать ее с результатами футбольных матчей, даже притом, что визуализация создавалась с прицелом на матчи с 10+ голами. Я загрузила футбольные данные, и визуализация отработала очень хорошо даже на матчах с 1–3 голами. Примерно в то же время я дополнила визуализацию результатами пенальти.



Визуализация накопления очков с результатами чемпионата Европы по футболу 2021 года

Но когда началась Олимпиада, я не смогла получить данные по матчам, потому что они были опубликованы в формате PDF. У меня получилось создать и опубликовать только две-три анимации, потому что извлечение данных из PDF было утомительной ручной работой, которую я не планировала.

Это научило меня искать лучшие источники данных. К счастью, другие события, такие как чемпионаты мира или состязания, проводимые спортивными ассоциациями, сопровождались более легкодоступными данными. Я создала намного больше таких анимаций после Олимпиады, когда у меня появились хорошие источники данных.

Вы создавали анимацию с помощью D3? Использовали ли при этом другие библиотеки или инструменты?

Я использовала только D3.js, но для публикации изображений я записывала анимации с экрана с помощью программы ScreenToGif.

У вас такой узнаваемый стиль! Как он развивался со временем? Есть ли какие-то хитрости по поиску «своего голоса» в области визуализации данных?

Основы я изучала в университете, но современные тенденции графического дизайна (печать, дизайн бренда, дизайн упаковки, типографика) тоже сильно влияют на мой стиль. Я ежедневно просматриваю сайты по графическому дизайну в поисках вдохновения. Когда я нахожу визуальный элемент или узор, который мне нравится, я сразу начинаю думать: «Это можно приспособить для визуализации данных».

Конечно, не каждая тенденция подходит для применения в диаграммах, но я многому учусь, пытаясь понять, как с помощью D3.js воссоздать тот или иной необычный вид.

Что бы вы посоветовали нашим читателями, изучающим D3 и занимающимся созданием своих проектов?

Разделите этапы создания дизайна и реализации. Сначала спроектируйте внешний вид, намеренно избегая мыслей о том, насколько сложно будет воплотить его. Так вы всегда будете сталкиваться с новыми задачами и узнаете много новых приемов.

Если дизайн не является вашей сильной стороной, то попробуйте поискать хорошую визуализацию, а затем просто воссоздайте ее с нуля.

Итоги

- Создавая визуализации, мы работаем с двумя основными типами данных: количественными и качественными. Количественные данные – это числовая информация, например вес или температура, а качественные – это, как правило, текстовая информация, например названия стран или жанры фильмов.
- Как разработчикам на D3 нам приходится работать с наборами данных в разных форматах. Наиболее распространены табличные форматы, такие как CSV и JSON. Но данные также могут быть организованы способами, подходящими для иерархической, сетевой или географической визуализации.
- D3 предлагает функции для загрузки в проект наборов данных определенных форматов. Например, функции `d3.csv()` и `d3.json()` могут загружать файлы CSV или JSON соответственно. По мере загрузки D3 преобразует данные в массив объектов.
- При загрузке внешних наборов обычно необходимо правильно отформатировать данные, особенно числа. Функция обратного вызова, которая передается в вызов `d3.csv()` и `d3.json()`, имеет доступ к набору данных построчно и может использоваться для преобразования типов и выполнения других манипуляций.

- Загрузка данных в проект – асинхронный процесс, т. е. во время загрузки данных браузер продолжает читать и выполнять сценарий, поэтому:
  - важно дождаться полного завершения загрузки данных, прежде чем начинать манипулировать ими. С этой целью можно использовать JavaScript-объекты `Promise` с методом `then()`;
  - функция обратного вызова, которая передается в вызов метода `then()`, получает доступ ко всему набору данных после его загрузки. В ней можно оценить и реорганизовать данные перед построением визуализации.
- Шаблон привязки данных генерирует столько элементов SVG, сколько имеется отдельных точек данных или строк в табличном наборе данных:
  - шаблон привязки данных состоит из трех методов, применяемых в цепочке к выборке: `selectAll()`, `data()` и `join()`;
  - после создания элементов SVG с использованием шаблона привязки данных мы можем обращаться к связанным данным через встроенные функции;
  - данные, привязанные к элементу, также передаются его дочерним элементам.
- Шкалы D3 позволяют преобразовывать значения из набора данных в атрибуты, применяемые к элементам SVG, такие как их размер, положение или цвет:
  - шкалы можно сгруппировать в четыре семейства в зависимости от дискретности или непрерывности их входных и выходных значений;
  - линейные шкалы принимают непрерывные входные данные и возвращают значения из непрерывного выходного диапазона. Выходные данные линейно пропорциональны входным данным. Линейные шкалы широко используются в проектах D3, например для вычисления длин прямоугольников в столбчатой диаграмме;
  - интервальные шкалы принимают дискретные входные данные и возвращают значения из непрерывного выходного диапазона. Они часто применяются для равномерного распределения прямоугольников по доступному пространству в столбчатой диаграмме.
- Операция переноса группы SVG влияет на все ее дочерние элементы.
- Метки создаются с помощью текстовых элементов SVG. Каждый текстовый элемент должен позиционироваться отдельно с использованием его атрибутов `x` и `y`. Также с помощью атрибута `text-anchor` можно задать выравнивание текста по правому краю.

4 Рисование линий и дуг

В этой главе рассматриваются:

- добавление осей в диаграмму и применение соглашений об отступах;
- рисование линейных графиков с помощью функции-генератора линий;
- интерполяция точек данных для рисования кривых;
- рисование областей с помощью генераторов областей;
- рисование дуги с помощью генератора дуг.

Вы уже знакомы с фигурами SVG, наиболее часто используемыми для визуализации данных: линиями, прямоугольниками и кругами. Мы даже создали столбчатую диаграмму с нуля, используя прямоугольники. Однако с помощью простых фигур можно нарисовать не так много. Для создания более сложных визуализаций мы обычно используем кривые SVG. Как уже говорилось в главе 1, кривые – это самые гибкие из элементов SVG, способные принимать практически любую форму. Они широко используются в проектах D3; простейшими примерами может служить рисование прямых и кривых линий в линейных графиках или дуг в кольцевых диаграммах.

Форма кривой SVG определяется ее атрибутом `d`. Этот атрибут содержит строку с последовательностью команд, определяющих на-

чальную и конечную точки кривой, типы кривых, которые используются для изменения направления, и признак замкнутости кривой. Атрибут d может быть очень длинным и сложным. Но в большинстве случаев мы не пишем его сами, а используем генераторы фигур, имеющиеся в D3!

В этой главе мы создадим визуализацию, изображенную на рис. 4.1: линейный график изменения температуры и набор дуг, визуализирующих процент дней с осадками в Нью-Йорке в 2021 году. Исходный код этой визуализации вы найдете по адресу http://mng.bz/5orB. Базовые данные взяты с сайта Weather Underground (www.wunderground.com).

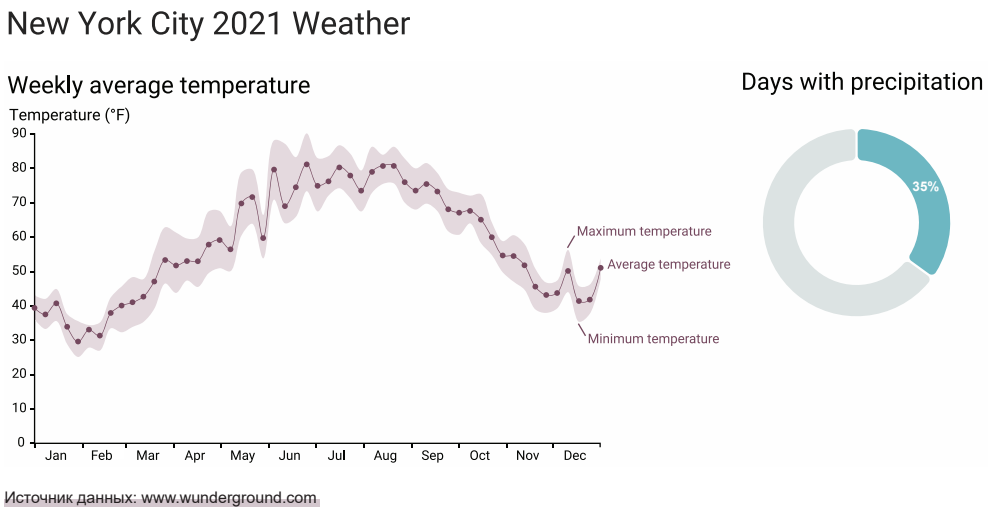


Рис. 4.1 Визуализация, которая будет построена в этой главе: линейная диаграмма изменения температуры в Нью-Йорке в 2021 году и набор дуг, показывающих процент дней с осадками

Обе диаграммы мы создадим с помощью функций-генераторов фигур из D3. Но прежде давайте обсудим соглашение о границах и отступах, используемое в D3, и узнаем, как добавлять оси в диаграммы.

4.1 Добавление осей

При создании визуализаций данных часто желательно заранее спланировать использование пространства контейнера SVG. Конечно, всегда хочется побыстрее начать эксперименты с крутыми возможностями – т. е. с ядром визуализации, – но поверьте нам, небольшая предварительная подготовка поможет вам сэкономить массу вре-

мени. Это верно для всех задач в программировании, да и вообще в жизни! На этапе предварительного планирования мы размышляем не только о самой диаграмме, но и о дополнительных элементах, делающих диаграмму более удобочитаемой, таких как оси, метки и легенды.

В этом разделе мы познакомимся с соглашением о границах и отступах, помогающим облегчить распределение пространства для различных элементов. Затем мы узнаем, как добавить оси в диаграмму, и познакомимся с несколькими элементами SVG, составляющими оси в D3. Эти новые знания мы применим для создания линейной диаграммы, показанной на рис. 4.1.

Для начала обратимся к файлам с примерами кода для главы 4. Их можно загрузить из репозитория книги на GitHub (http://mng.bz/Xqjv). В папке `chapter_04` файлы кода организованы по разделам. Чтобы приступить к выполнению упражнений в этой главе, откройте папку `4.1-Margin_convention_and_axes/start` в редакторе и запустите локальный веб-сервер. Если вам нужна помощь в настройке локальной среды разработки, то обратитесь к приложению A. Более подробную информацию о структуре папок проекта вы найдете в файле `README`.

> **ВНИМАНИЕ** При работе с примерами кода для этой главы открывайте в редакторе только одну папку, `start` или `end`. Если открыть сразу обе и использовать расширение Live Server для обслуживания проекта, то некоторые пути не будут работать так, как ожидалось.

Начнем с файла `line-chart.js` и загрузим набор данных о средних температурах за неделю с помощью метода `d3.csv()`:

```
d3.csv("../data/weekly_temperature.csv");
```

В главе 3 мы объяснили, что преобразования, выполняемые библиотекой D3 при загрузке табличного набора данных, могут влиять на типы значений. Например, числа в исходном наборе данных могут преобразовываться в строки, и нам нужно преобразовать их обратно в числа, чтобы облегчить последующие манипуляции. Мы видели, что функция обратного вызова, передаваемая в `d3.csv()`, имеет доступ к записям с данными и является отличным местом для выполнения таких преобразований. Здесь мы покажем небольшой трюк: вместо преобразования числа вручную мы вызовем метод `d3.autoType`. Он обнаруживает наиболее распространенные типы данных, такие как даты и числа, и преобразует их в соответствующий тип JavaScript:

```
d3.csv("../data/weekly_temperature.csv", d3.autoType);
```

Имейте в виду, что типы данных могут быть неоднозначными и `d3.autoType` иногда выбирает неправильный тип. Например, четырехзначные числа, такие как `2023`, могут обозначать как числа, так и даты. `d3.autoType` приведет такие значения к числовому типу, но это может быть не тем, что вам нужно. Поэтому всегда дважды проверяйте массив данных после полной загрузки. На Observable, кстати, есть отличная статья на эту тему (https://observablehq.com/@d3/d3-autotype).

В следующем фрагменте показано, как получить доступ к загруженному набору данных с помощью JavaScript-объекта `Promise` и вывести его в консоль, чтобы проверить правильность форматирования дат в виде дат JavaScript, а температур – в виде чисел. Результат можно увидеть на рис. 4.2:

```
d3.csv("../data/weekly_temperature.csv", d3.autoType).then(data => {
    console.log("temperature data", data);
});
```

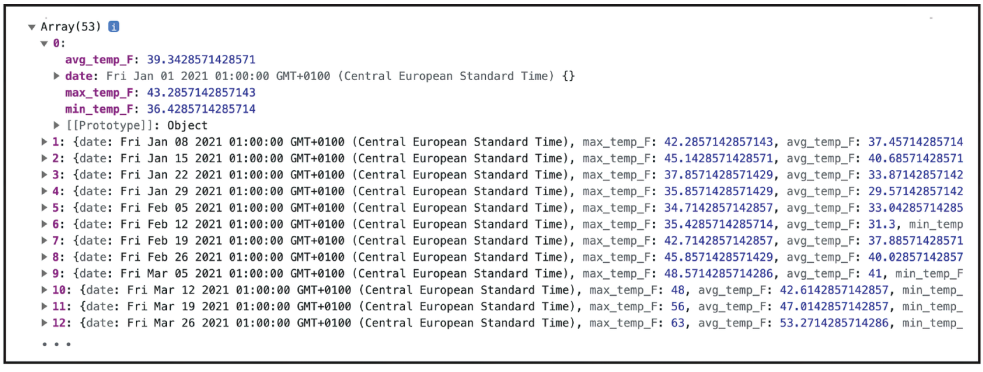


Рис. 4.2 Метод `d3.autoType` отформатировал даты как даты JavaScript, а температуры – как числа

Мы использовали JavaScript-объект `Promise`, потому что загрузка данных – асинхронный процесс (обратитесь к главе 3, чтобы вспомнить, как производятся загрузка и доступ к данным с помощью D3). Но теперь, зная, что набор данных полностью загружен и правильно отформатирован, можно начинать строить диаграмму.

Файл `line-chart.js` уже содержит функцию `drawLineChart()`, в которой мы создадим линейную диаграмму. В функции обратного вызова, что передается объекту `Promise`, вызовем функцию `drawLineChart()` и передадим ей загруженный набор данных:

```
d3.csv("../data/weekly_temperature.csv", d3.autoType).then(data => {
  console.log("temperature data", data);
  drawLineChart(data);
});
```

Теперь перейдем к обсуждению соглашения о границах и отступах и применим его к нашему графику!

4.1.1 Соглашение о границах и отступах

Соглашение о границах и отступах, принятое в D3, определяет общий порядок резервирования пространства вокруг диаграммы для осей, меток и легенд. В этом соглашении определяются четыре поля отступов: сверху, справа, снизу и слева от диаграммы, как показано на рис. 4.3. Задавая эти поля, можно узнать местоположение и размер области, оставшейся для ядра диаграммы, которое мы будем называть *внутренней диаграммой*.

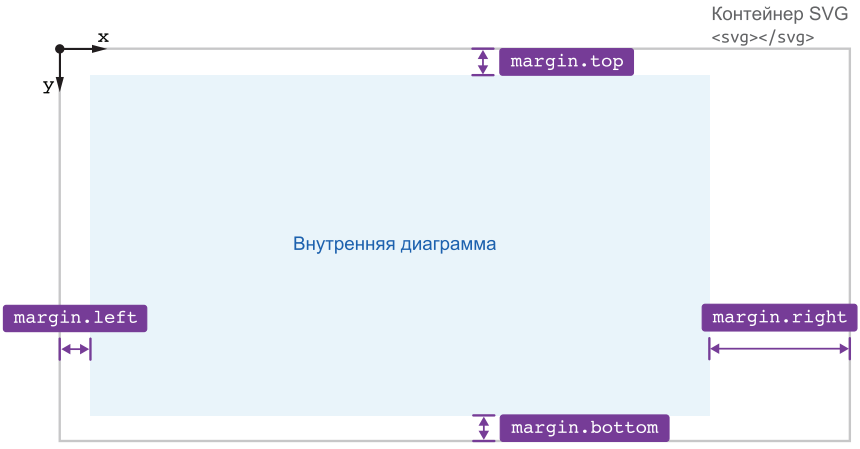


Рис. 4.3 Соглашение о границах и отступах D3 устанавливает значения отступов сверху, справа, снизу и слева от диаграммы

Значения отступов объявляются в объекте с атрибутами `top`, `right`, `bottom` и `left`. Давайте создадим такой объект для нашей линейной диаграммы. Внутри функции `drawLineChart()` объявим константу с именем `margin` и присвоим атрибутам `top`, `right`, `bottom` и `left` значения `40`, `170`, `25` и `40` соответственно:

```
const drawLineChart = (data) => {
  const margin = {top: 40, right: 170, bottom: 25, left: 40};
};
```

Обычно заранее точно неизвестно, сколько места понадобится для осей и меток. Поэтому сначала мы исходим из некоторого разумного предположения, а позже корректируем размеры отступов, если понадобится. Например, посмотрите на линейный график на рис. 4.1 или на онлайн-версию проекта (http://mng.bz/5orB): метки, отображаемые в правой части, относительно длинные, поэтому для поля

справа выделено 170 пикселей. С другой стороны, метки осей не занимают много места, поэтому остальные поля могут быть намного меньше.

После объявления объекта с отступами можно подумать о размере контейнера SVG. Зная размер контейнера SVG и полей, можно вычислить две новые константы – `innerWidth` и `innerHeight`, – представляющие ширину и высоту внутренней диаграммы. Эти размеры показаны на рис. 4.4.

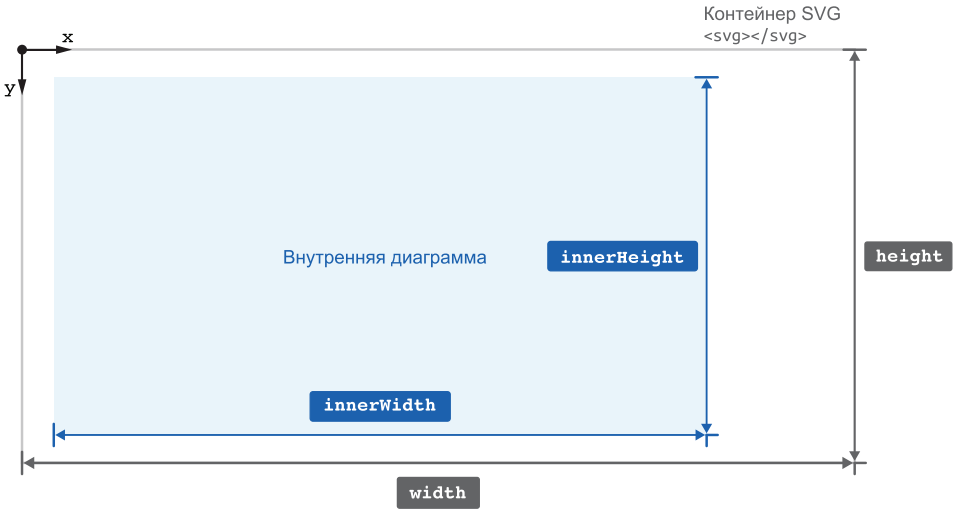


Рис. 4.4 Зная размеры контейнера SVG и полей, можно вычислить ширину и высоту внутренней диаграммы

Ширина внутренней диаграммы соответствует ширине контейнера SVG за вычетом полей слева и справа. Если контейнер SVG имеет ширину 1000 пикселей и определены правое и левое поле с размерами 170 и 40 пикселей соответственно, то для внутренней диаграммы остается пространство шириной 790 пикселей. Аналогично если высота контейнера SVG составляет 500 пикселей, то высоту внутренней диаграммы можно вычислить, отняв верхнее и нижнее поля от общей высоты, откуда получаем 435 пикселей. Сделав константы `innerWidth` и `innerHeight` пропорциональными полям, мы гарантируем, что они будут автоматически корректироваться, если позже понадобится изменить размеры полей:

```
const margin = {top: 40, right: 170, bottom: 25, left: 40};
const width = 1000;
const height = 500;
const innerWidth = width - margin.left - margin.right;
const innerHeight = height - margin.top - margin.bottom;
```

Теперь добавим контейнер SVG для нашей линейной диаграммы. Внутри все той же функции `drawLineChart()` добавим элемент SVG в `div` с идентификатором `line-chart`, который уже существует в файле `index.html`, и настроим его атрибут `viewBox`, используя константы `width` и `height`. Дополнительно можно временно включить рамку в элементе SVG, чтобы лучше видеть рабочую область (подробно о добавлении элементов в DOM и настройке их атрибутов и стилей рассказывается в главе 2):

```
const svg = d3.select("#line-chart")
  .append("svg")
  .attr("viewBox", `0, 0, ${width}, ${height}`);
```

Выше мы определили размеры отступов, которые будут определять область, зарезервированную для внутренней диаграммы. Зная, что система координат контейнера SVG начинается в верхнем левом углу, каждый элемент внутренней диаграммы нужно сместить в сторону зарезервированной области. Но чтобы не применять это смещение к каждому элементу, можно заключить внутреннюю диаграмму в группу SVG и применить смещение только к этой группе. Как можно видеть на рис. 4.5, эта стратегия создает новую систему координат для внутренней диаграммы.

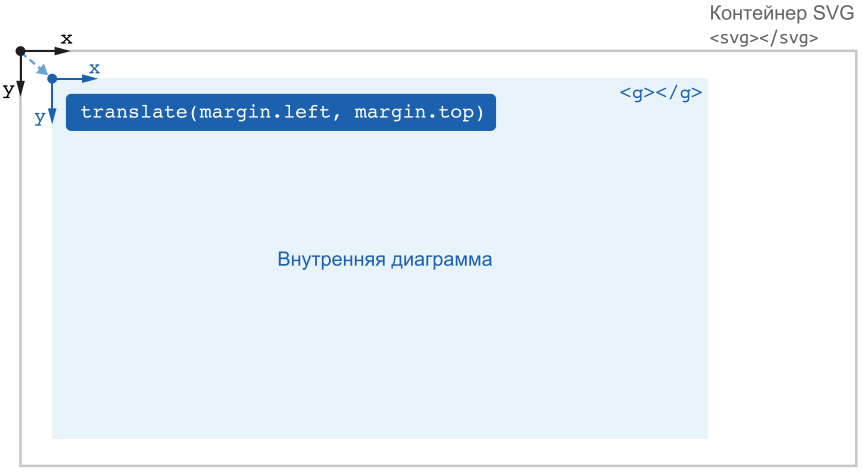


Рис. 4.5 Смещение группы SVG, которая будет содержать внутреннюю диаграмму, создает новую систему координат для элементов во внутренней диаграмме

Чтобы реализовать эту стратегию, добавим группу в контейнер SVG, применим к ней операцию переноса, опираясь на размеры отступов слева и сверху, и сохраним группу в константу с именем `innerChart`, которую позднее будем использовать для построения линейной диаграммы:

```
const innerChart = svg
  .append("g")
  .attr("transform", `translate(${margin.left}, ${margin.top})`);
```

Главное преимущество соглашения о границах и отступах и представленной здесь стратегии – после ее реализации о ней больше не нужно помнить. Можно продолжать создавать оси и диаграмму, зная, что области для меток, легенд и другой дополнительной информации уже зарезервированы.

4.1.2 Генерирование осей

Настроив отступы в соответствии с соглашением, можно добавить оси в диаграмму. Оси – неотъемлемая часть большинства визуализаций данных. Они играют роль справки для зрителя, чтобы тот мог понять представленные числа и категории.

На линейном графике на рис. 4.1 или в онлайн-версии проекта (http://mng.bz/5orB) можно заметить две оси. Горизонтальная ось, также называемая осью `x`, показывает положение каждого месяца. Вертикальная ось, или ось `y`, служит для представления температуры в градусах Фаренгейта.

В D3 оси создаются с помощью генератора компонентов `axis()`. Он принимает шкалу и возвращает элементы SVG, составляющие ось. Обсуждая шкалы в главе 3, мы говорили, что они преобразуют значения из предметной области в значения из заданного диапазона, которые затем можно использовать для рисования на экране. Соответственно, определив шкалы для нашей линейной диаграммы, мы сможем автоматически получать горизонтальное местоположение каждой даты из набора данных или вертикальное местоположение связанных с ними температур.

Объявление шкал

Первый шаг к созданию осей – объявление соответствующих шкал. Прежде всего нам нужна шкала, располагающая даты по горизонтали. Это действие выполняет `d3.scaleTime()` – шкала времени в D3 (см. приложение B, где рассказывается, как выбрать правильную шкалу). Шкала времени относится к первому семейству шкал, обсуждавшихся в главе 3. Она принимает и возвращает непрерывные данные. Шкала времени очень похожа на линейную шкалу, использовавшуюся в главе 3, с той лишь разницей, что манипулирует данными, связанными со временем.

Давайте дадим шкале времени имя `xScale`, поскольку она будет отвечать за позиционирование элементов по оси `x`. Предметная область шкалы простирается от первой до последней даты в наборе данных. В следующем фрагменте мы используем методы `d3.min()` и `d3.max()` из модуля d3-array, чтобы найти эти значения.

Диапазон шкалы простирается вдоль доступного горизонтального пространства во внутренней диаграмме (рис. 4.5), т. е. от 0 до `innerWidth` (об объявлении предметной области и диапазона в D3 рассказывается в главе 3):

```
const firstDate = d3.min(data, d => d.date);
const lastDate = d3.max(data, d => d.date);
const xScale = d3.scaleTime()
  .domain([firstDate, lastDate])
  .range([0, innerWidth]);
```

Для распределения температур вдоль оси `y` тоже нужна шкала с непрерывными входом и выходом. В данном случае идеально подходит линейная шкала, поскольку нам нужно, чтобы температура и вертикальное положение на линейной диаграмме были линейно пропорциональны.

Следующий фрагмент объявляет температурную шкалу с именем `yScale`, потому что она отвечает за позиционирование элементов вдоль оси `y`. Наша ось `y` должна начинаться с 0, поэтому в качестве минимального значения предметной области мы передаем 0. Даже притом что минимальная температура в наборе данных составляет около 26 °F, начать ось `y` с 0 – хорошая идея и, в нашем случае, позволит правильно передать динамику изменения температуры. Но, как и многое в жизни, это не жесткое правило, и для этой диаграммы нет правильного или неправильного ответа, особенно потому, что 0 по шкале Фаренгейта не является абсолютным 0.

В качестве максимального предметного значения мы передаем максимальную температуру из набора данных. Это значение извлекается из столбца `max_temp_F` в наборе данных с помощью функции `d3.max()`.

Диапазон шкалы простирается вдоль доступного вертикального пространства высоты во внутренней диаграмме, а поскольку вертикальные значения вычисляются в системе координат SVG сверху вниз, диапазон начинается с `innerHeight` (позиции нижнего левого угла внутренней диаграммы) и заканчивается 0 (позицией верхнего левого угла):

```
const maxTemp = d3.max(data, d => d.max_temp_F);
const yScale = d3.scaleLinear()
  .domain([0, maxTemp])
  .range([innerHeight, 0]);
```

Добавление осей

После инициализации шкал можно добавить оси. В D3 есть четыре генератора осей – `axisTop()`, `axisRight()`, `axisBottom()` и `axisLeft()`, – которые создают компоненты верхней, правой, нижней и левой осей соответственно. Все они являются частью модуля d3-axis.

Мы уже упоминали, что функции-генераторы осей принимают шкалу на входе. Например, чтобы создать нижнюю ось для нашей диаграммы, нужно вызвать генератор `axisBottom()` и передать ему шкалу `xScale`, потому что она отвечает за распределение данных вдоль нижней оси. Сохраним результат работы генератора в константе с именем `bottomAxis`:

```
const bottomAxis = d3.axisBottom(xScale);
```

Генератор осей – это функция, создающая элементы, которые составляют ось. Чтобы эти элементы появились на экране, нужно применить генератор осей к выборке D3 с помощью метода `call()`. Обратите внимание, что в следующем фрагменте используется выборка `innerChart`, в которую перед вызовом генератора осей добавляется элемент группы. Группа имеет класс с именем `axis-x`, что позже поможет позиционировать и оформить внешний вид оси:

```
const bottomAxis = d3.axisBottom(xScale);

innerChart
  .append("g")
    .attr("class", "axis-x")
    .call(bottomAxis);
```

Взгляните на сгенерированную ось в браузере. По умолчанию оси D3 отображаются в начале системы координат выборки – в данном случае в верхнем левом углу внутренней области диаграммы (рис. 4.6). Чтобы переместить ось вниз, нужно сместить группу SVG, содержащую эту ось. Напомним, что преобразования, применяемые к группе, наследуются всеми ее элементами. В следующем фрагменте мы смещаем группу, содержащую элементы оси, вниз на величину высоты внутренней диаграммы:

```
const bottomAxis = d3.axisBottom(xScale);
innerChart
  .append("g")
    .attr("class", "axis-x")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxis);
```

Также нам нужно изменить форматирование меток осей. По умолчанию D3 адаптирует представление времени на осях, отображая метки часов, дней, месяцев или лет, опираясь на заданную предметную область. Но форматирование по умолчанию не всегда дает устраивающие нас метки. К счастью, D3 предлагает несколько методов для изменения формата меток.

Прежде всего отметим, что ось x имеет метки для месяцев с февраля по декабрь, что очень хорошо, но на ней нет метки для января. В зависимости от часового пояса, в котором вы проживаете, первая дата может не совпадать с полночью первого января, что не позволяет D3 распознать ее как начало первого месяца. Поскольку наш набор данных не является динамическим, мы можем позволить себе жестко зафиксировать переменную `firstDate`. Для этого воспользуемся конструктором `Date()`.



Рис. 4.6 По умолчанию оси D3 отображаются в начале системы координат выборки – в данном случае в верхнем левом углу внутренней области диаграммы. Мы должны сместить их в нужное место

Следующий фрагмент создает `firstDate` – новый объект `Date()`. В скобках сначала объявляется год (`2021`), месяц (`00`, потому что нумерация месяцев начинается с нуля) и день (`01`), а затем, если необходимо, можно указать часы, минуты и секунды (`0, 0, 0`):

```
const firstDate = new Date(2021, 00, 01, 0, 0, 0);
const lastDate = d3.max(data, d => d.date);
const xScale = d3.scaleTime()
  .domain([firstDate, lastDate])
  .range([0, innerWidth]);
```

Сохранив проект, вы увидите, что теперь в месте, соответствующем 1 января, появилась метка. Но метка сообщает только год 2021, как показано на рис. 4.7. Это не неправильно, учитывая, что 00:00:00 пятница 1 января 2021 соответствует самому началу 2021 года, но мы бы предпочли иметь на этом месте метку месяца.



Рис. 4.7 По умолчанию D3 адаптирует представление времени на метках осей. В нашем случае она представляет 1 января как начало 2021 года. Это не неправильно, но и не идеально для читабельности

При желании формат меток при рисках на осях можно изменить с помощью метода `axis.tickFormat()` из модуля d3-axis. *Риски* – это короткие черточки на оси. Они часто, но не всегда, сопровождаются метками.

Допустим, мы решили вывести при рисках сокращенные названия месяцев. В D3 значения даты и времени можно форматировать с помощью метода `d3.timeFormat()` из модуля d3-time-format. Этот метод принимает параметр со спецификатором формата, например `%b`, который выводит сокращенные названия месяцев. Полный список доступных спецификаторов можно увидеть в модуле.

В следующем фрагменте мы добавляем метод `tickFormat()` в цепочку создания нижней оси и передаем ему формат вывода времени:

```
const bottomAxis = d3.axisBottom(xScale)
  .tickFormat(d3.timeFormat("%b"));
```

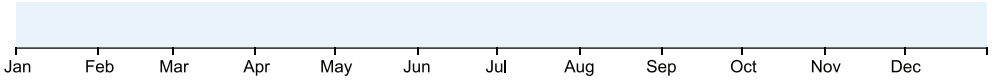


Рис. 4.8 Метки нижней оси в формате представления сокращенных названий месяцев

Теперь метки отформатированы правильно! Они отмечают начало каждого месяца, и это уже хорошо, но мы могли бы улучшить читаемость, разместив метки месяцев по центру между соответствующими им рисками, чтобы было понятно, что каждый месяц простирается от одной риски до следующей.

Чтобы изменить местоположение меток, их сначала нужно выбрать. Откройте панель инспектора в браузере и внимательно посмотрите на элементы SVG, сгенерированные библиотекой D3 для создания оси. Во-первых, среди них есть элемент `path` с классом `domain`, который рисует горизонтальную линию, охватывающую диапазон предметных значений, которая включает две внешние риски на концах, как можно видеть на рис. 4.9. Риски и метки оси состоят из прямой линии и текстового элемента, объединенных в группы SVG с классом `tick`. Эти группы смещаются вдоль оси для установки в правильные местоположения. Типы и классы элементов, созданных генератором оси, являются частью открытого D3 API. Их можно использовать для настройки внешнего вида оси.



Рис. 4.9 Элементы SVG, составляющие ось

Учитывая такую организацию, можно выбрать все метки оси x с помощью селектора ".axis-x text" (соответствует всем текстовым элементам в группе с классом axis-x) и выполнить нужные корректировки. Сначала переместим текстовые элементы вниз на 10 пикселей, используя их атрибут y. Это увеличенное пустое пространство по вертикали улучшит удобочитаемость. Также установим их свойство font-family в значение "Roboto" (этот шрифт мы уже использовали в проекте). По умолчанию D3 настраивает свойство font-family элемента axis, присваивая ему значение "sans-serif", не позволяя меткам наследовать свойство font-family из корня проекта. Наконец, увеличим font-size до 14 пикселей.

Если следовать принципу отделения логики от представления, то эти последние две корректировки свойств стиля желательно переместить в файл CSS. Здесь мы делаем это с помощью D3, только чтобы упростить объяснения:

```
d3.selectAll(".axis-x text")
  .attr("y", "10px")
  .style("font-family", "Roboto, sans-serif")
  .style("font-size", "14px");
```

Чтобы расположить метки с названиями месяцев посередине между соответствующими им рисками, используем атрибут x. Поскольку каждый месяц имеет разную продолжительность (от 28 до 31 дня), для каждой метки нам нужно найти середину между первым днем текущего месяца и первым днем следующего месяца. Обратите внимание, что D3 уже установила в g.axis-x свойство text-anchor в значение "middle".

Мы знаем, что данные, привязанные к каждой метке, соответствуют первому дню месяца. Следующий фрагмент отыскивает следующий месяц вызовом JavaScript-метода getMonth() текущего месяца (значения, привязанного к метке). Он возвращает число от 0 до 11, где 0 соответствует январю, а 11 – декабрю. Затем можно создать новую дату, передав год, следующий месяц и первый день месяца объекту Date(). Как упоминалось в главе 3, первый параметр, передаваемый в функцию обратного вызова, который часто называют d, соответствует каждой строке из связанного набора данных. Это напоминает обход элементов массива с помощью метода forEach().

После этого остается только вычислить половину расстояния между началом текущего и началом следующего месяца с помощью

xScale. После завершения ось y вас должна выглядеть, как показано на рис. 4.10:

```
d3.selectAll(".axis-x text")
  .attr("x", d => {
    const currentMonth = d;
    const nextMonth = new Date(2021, currentMonth.getMonth() + 1, 1);
    return (xScale(nextMonth) - xScale(currentMonth)) / 2;
  })
  .attr("y", "10px")
  .style("font-family", "Roboto, sans-serif")
  .style("font-size", "14px");
```



Рис. 4.10 Отформатированная ось x с метками месяцев, расположенными по центру между соответствующими рисками

Мы проделали довольно много манипуляций, но они должны дать вам общее представление о различных способах настройки осей D3.

Теперь добавим ось y. Для этого используем генератор осей d3.axisLeft(), потому что ось y должна располагаться на левой стороне диаграммы. Передадим генератору yScale и сохраним ось в константе с именем leftAxis:

```
const leftAxis = d3.axisLeft(yScale);
```

Как и прежде, добавим ось во внутреннюю диаграмму, для этого вставим группу в конец выборки с внутренней диаграммой, присвоим ей класс axis-y и вызовем leftAxis:

```
const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
  .attr("class", "axis-y")
  .call(leftAxis);
```

Сохранив проект и открыв его в браузере, вы увидите, что ось y уже расположена правильно (рис. 4.11). Нам осталось лишь изменить шрифт меток. Следующий фрагмент выбирает все текстовые элементы внутри группы с классом axis-y, смещает их влево для лучшей читаемости с помощью атрибута x и изменяет их свойства font-family и font-size:

```
d3.selectAll(".axis-y text")
  .attr("x", "-5px")
  .style("font-family", "Roboto, sans-serif")
  .style("font-size", "14px");
```



Рис. 4.11 Окончательно настроенные оси x и y

Возможно, вы заметили повторяющийся код, который настраивает свойства `font-family` и `font-size` меток двух осей. Для обучения это не имеет большого значения, но в своей профессиональной практике мы стараемся избегать такого повторения. Лучшее решение, как уже говорилось выше, – определить стили в файле CSS. Другое возможное решение – применить их с помощью комбинированного селектора, например:

```
d3.selectAll(".axis-x text, .axis-y text")
  .style("font-family", "Roboto, sans-serif")
  .style("font-size", "14px");
```

Добавление меток осей

Мы почти закончили оси, осталось лишь помочь читателям понять нашу диаграмму. Метки делений на оси x говорят сами за себя, а вот на оси y – нет. Мы видим, что значения по оси y изменяются в диапазоне от 0 до 90, но понятия не имеем, что это за значения.

Этот недостаток легко исправить, добавив метку оси. В проектах D3 метки – это простые текстовые элементы, поэтому нужно лишь добавить текстовый элемент в контейнер SVG. Присвоим ему значение `"Температура (°F)"` и сместим на 20 пикселей ниже начала координат контейнера SVG:

```
svg
  .append("text")
  .text("Temperature (°F)")
  .attr("y", 20);
```

Вот и все! Теперь проект должен выглядеть как на рис. 4.12. В следующем разделе мы нарисуем линейную диаграмму.



Рис. 4.12 Окончательный вид осей и меток

4.2 Построение линейной диаграммы

Теперь можно приступать к построению одной из самых распространенных визуализаций данных: линейной диаграммы. Линейные диаграммы состоят из линий, соединяющих точки данных, или кривых, интерполирующих эти точки. Они часто используются, чтобы показать, как развиваются явления с течением времени. В D3 эти линии и кривые строятся с помощью SVG-элементов `path`, формы которых определяются их атрибутами `d`. В главе 1 мы видели, что атрибут `d` определяется как строка с серией команд, которые определяют порядок рисования фигуры. Мы также говорили, что эта строка может быть очень сложной. К счастью, модуль d3-shape предоставляет функции-генераторы линий и кривых, которые помогают заполнить атрибут `d` и облегчают создание линейных диаграмм.

В этом разделе мы нарисуем кривую, показывающую изменение средней температуры в Нью-Йорке в течение 2021 года, как можно видеть в онлайн-версии проекта (http://mng.bz/5orB) или на рис. 4.1. Но прежде отобразим каждую точку данных. Этот шаг не является строго необходимым для построения линейной диаграммы, но он поможет вам понять, как работает генератор линий D3.

Внутри функции `drawLineChart()` используем связанные данные и нарисуем круг для каждой записи в наборе данных `weekly_temperature.csv`. Для этого добавим круги в выборку `innerChart`, зададим им радиус 4 пикселя и вычислим их атрибуты местоположения (`cx` и `cy`) с использованием шкал.

Как уже говорилось в главе 3, данные, привязанные к каждому кругу, можно получить с помощью функции доступа. В следующем фрагменте параметр `d` открывает доступ к записи, привязанной к текущему кругу. Поскольку данные представлены объектами JavaScript, мы можем получить дату или среднюю температуру с помощью точечной нотации. Обратитесь к разделу 3.3.1, если вам нужно освежить в памяти эту концепцию.

Обратите внимание, что мы объявили отдельную константу `aubergine`, определяющую цвет (баклажан), и использовали ее для установки атрибута `fill` кругов. В этом проекте мы используем один и тот же цвет несколько раз, поэтому наличие константы даст нам определенное удобство. При желании вы можете использовать любой другой цвет по своему выбору:

```
const aubergine = "#75485E";
innerChart
  .selectAll("circle")
  .data(data)
  .join("circle")
    .attr("r", 4)
    .attr("cx", d => xScale(d.date))
    .attr("cy", d => yScale(d.avg_temp_F))
    .attr("fill", aubergine);
```

Использование шаблона привязки данных, чтобы добавить в диаграмму круг для каждой записи в наборе данных

Использовать шкалы для позиционирования кругов на основе связанных с ними данных

Сохраните проект и посмотрите, как он выглядит в браузере. Точки данных должны быть расположены между 29 °F и 80 °F и иметь куполообразную форму, как на рис. 4.13.

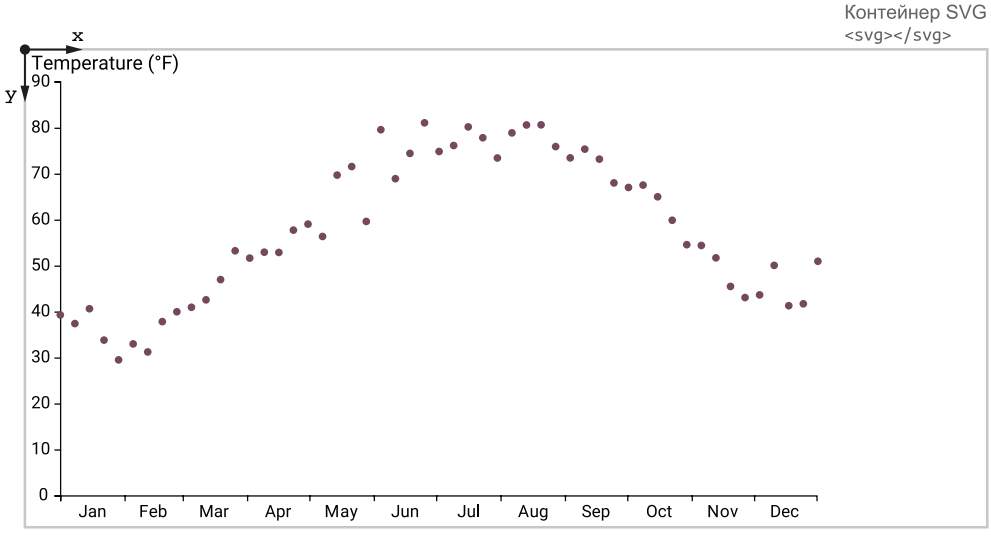
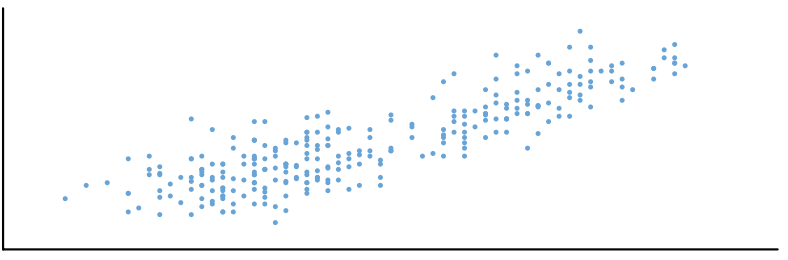


Рис. 4.13 Точки данных, отражающие изменение средней температуры с течением времени

Теперь вы умеете рисовать диаграммы рассеяния

Самое интересное, что, даже не заметив этого, вы научились рисовать диаграммы рассеяния! Диаграмма рассеяния – это просто диаграмма, показывающая набор точек данных в координатной плоскости и визуализирующая взаимосвязь между двумя или более переменными.

Вы узнали, как рисовать оси и как рисовать точки на экране на основе связанных данных, так что теперь вы умеете строить диаграммы рассеяния! Это одна из самых замечательных особенностей D3. Вам не нужно учиться создавать специальные диаграммы, достаточно научиться генерировать строительные блоки и собирать их вместе. В диаграммах рассеяния используются такие простые строительные блоки, как оси и круги. В главе 7 мы построим диаграмму рассеяния, где площадь кругов будет зависеть от значения переменной.



Пример диаграммы рассеяния

4.2.1 Использование генератора линий

Теперь, имея точки данных перед глазами, будет проще понять работу генератора линий в D3. Генератор линий `d3.line()` – это функция, принимающая горизонтальную и вертикальную координаты каждой точки данных и возвращающая атрибут `d` для SVG-элемента кривой или ломаной, проходящей через эти точки. Обычно в одну цепочку с генератором линий мы добавляем две функции доступа, `x()` и `y()`, которые принимают горизонтальную и вертикальную координаты точки данных соответственно (рис. 4.14).

Давайте определим функцию генератора линий для нашей линейной диаграммы. Сначала вызовем метод `d3.line()` и объединим этот вызов с функциями доступа `x()` и `y()`. Функция `x()` принимает горизонтальную координату каждой точки данных. Выполняя обход данных, как мы делали это до сих пор, можно использовать параметр `d`, дающий доступ к каждой отдельной точке данных. Горизонтальная координата каждой точки соответствует дате, которую она представляет, и рассчитывается с помощью `xScale()`. Аналогично

вертикальная координата точек данных пропорциональна средней температуре в этот день и вычисляется с помощью `yScale()`. Сохраним функцию генератора линии в константе с именем `lineGenerator`, чтобы иметь возможность вызвать ее позже:

```
const lineGenerator = d3.line()
  .x(d => xScale(d.date))
  .y(d => yScale(d.avg_temp_F));
```

Горизонтальная координата каждой точки данных

Вертикальная координата каждой точки данных

```
d3.line()
  .x( d => xScale(d.xValue) )
  .y( d => yScale(d.yValue) );
```

Горизонтальная координата каждой точки данных

Вертикальная координата каждой точки данных

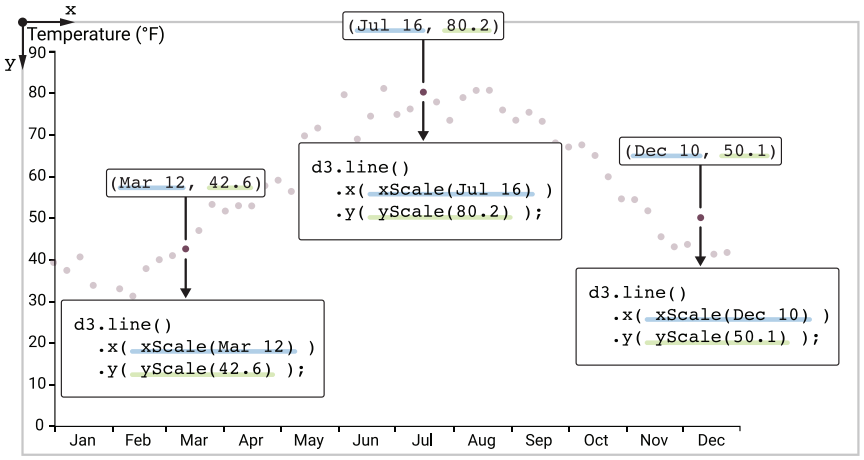


Рис. 4.14 Генератор линий `d3.line()` используется в сочетании с двумя функциями доступа, `x()` и `y()`, которые принимают горизонтальную и вертикальную координаты соответственно каждой точки данных

Затем добавим элемент `path` во внутреннюю диаграмму и зададим его атрибут `d`, вызвав генератор линий и передав ему набор данных.

По умолчанию кривые SVG имеют черную заливку. Чтобы нарисовать только линию, нужно присвоить атрибуту `fill` значение `none` или `transparent`, а атрибуту `stroke` – значение цвета; в данном случае мы присвоим цвет, который сохранили в константе `aubergine`. Теперь линия является частью нашей линейной диаграммы, как можно видеть на рис. 4.15:

```
innerChart
  .append("path")
    .attr("d", lineGenerator(data))
    .attr("fill", "none")
    .attr("stroke", aubergine);
```

Вызов генератора линий и передача ему набора данных

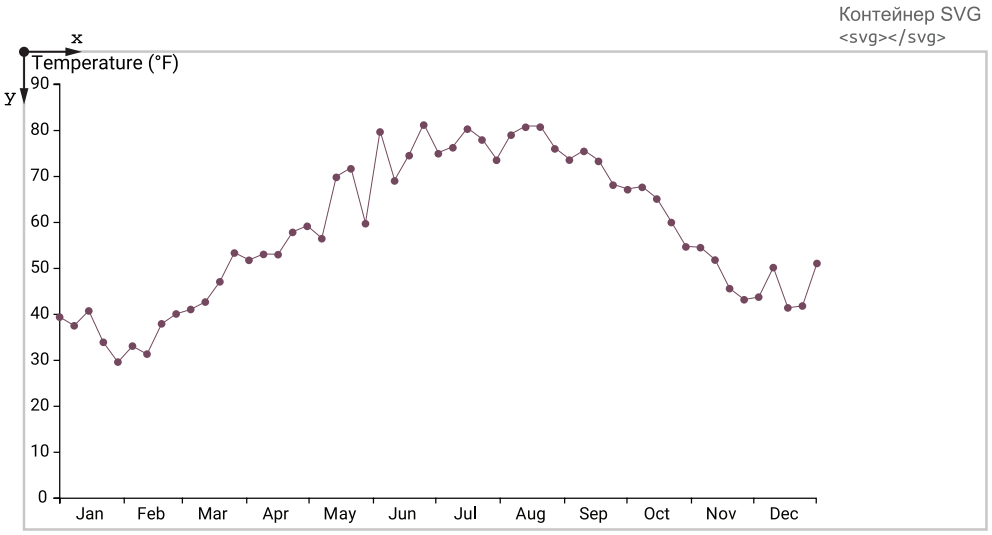


Рис. 4.15 Кривая SVG, созданная с помощью генератора линий, которая проходит через каждую точку данных и образует линейный график

4.2.2 Интерполяция точек данных в кривую

В случаях, подобных нашему линейному графику, где дискретные точки данных охватывают весь спектр данных, хорошим решением может быть представление точек данных простой линией. Но иногда нужно интерполировать данные между точками, для чего D3 предоставляет различные функции, генерирующие кривые.

Генераторы кривых используются как функция доступа в `d3.line()`. Чтобы преобразовать генератор линии, объявленный в предыдущем разделе, в генератор кривой, достаточно добавить в цепочку функцию доступа `curve()` и передать ей один из интерполяторов D3. В следующем фрагменте используется интерполятор `d3.curveCatmullRom`, который создает кубический сплайн (гладкую и гибкую кривую, проходящую через каждую точку данных и вычисляемую с помощью полиномиальной функции третьего порядка). Результат работы этого кода показан на рис. 4.16:

```
const curveGenerator = d3.line()
  .x(d => xScale(d.date))
  .y(d => yScale(d.avg_temp_F)
  .curve(d3.curveCatmullRom);
```

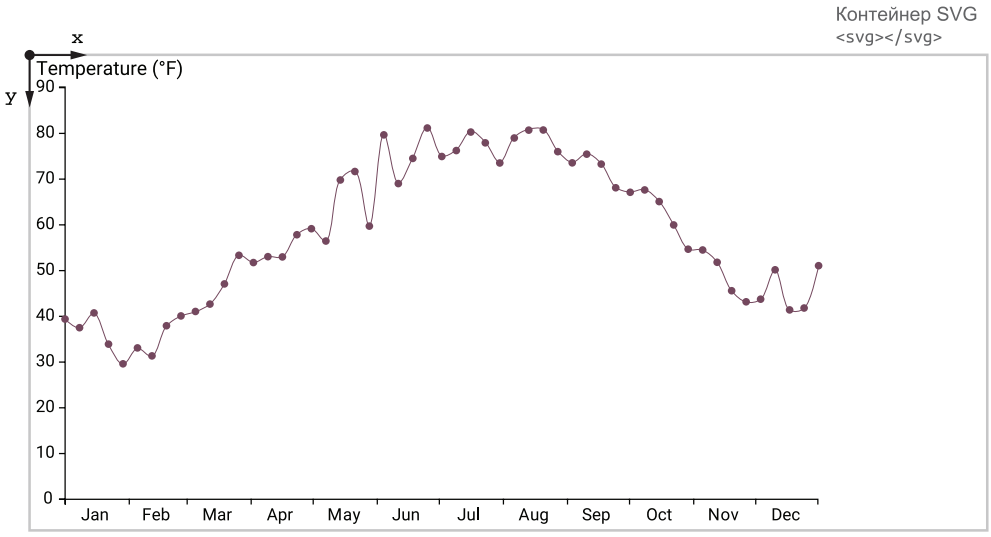


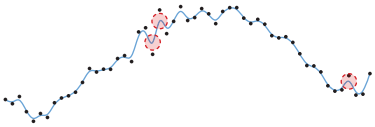
Рис. 4.16 Линейный график с интерполяцией сплайном Кэтмелла–Рома

Какой вид интерполяции лучше?

Интерполяция изменяет представление данных, и разные функции интерполяции создают разные представления. Данные можно визуализировать разными способами, ни один из которых нельзя назвать неправильным. Но при этом важно убедиться, что визуальное представление информации отражает реальные явления.

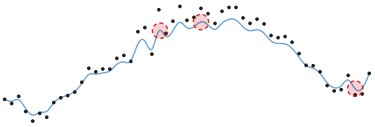
Поскольку визуализация данных имеет дело с визуальным представлением статистических принципов, она подвержена опасностям неправильного использования. Интерполяция линий особенно уязвима в этом отношении, поскольку превращает неприглядную ломаную в гладкую, «естественную» кривую.

На рис. 4.17 показан один и тот же линейный график, полученный с применением разных функций интерполяции, чтобы вы могли оценить, как они влияют на визуальное представление. Выбор адекватной функции интерполяции во многом зависит от данных, с которыми вы работаете. В нашем случае `d3.curveBasis` недооценивает внезапные изменения температуры, тогда как `d3.curveBundle` спрямляет кривую и уменьшает ее вариации, что неадекватно для наших данных. Если бы мы не нарисовали точки данных на графике, то не узнали бы, что кривая отображает их неточно. Вот почему важно тщательно выбирать и тестировать функцию интерполяции кривой.



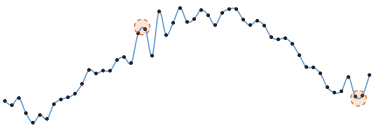
`d3.curveBasis`

Интерполирует простым кубическим сплайном, который проходит через первую и последнюю точки данных



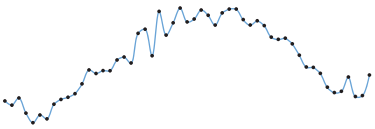
`d3.curveBundle`

Интерполирует простым сглаживающим кубическим сплайном



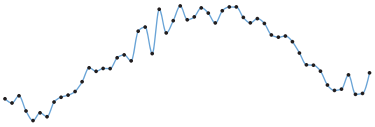
`d3.curveCardinal`

Интерполирует кардинальным сплайном, который проходит через каждую точку данных



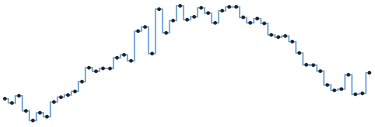
`d3.curveCatmullRom`

Интерполирует кубическим сплайном Кэтмелла–Рома, который проходит через каждую точку данных



`d3.curveMonotoneX`

Интерполирует кубическим сплайном, который проходит через каждую точку данных, сохраняя монотонность в вертикальном направлении



`d3.curveStep`

Интерполирует ступенчатой функцией, которая проходит через каждую точку данных и имеет вид чередующихся горизонтальных и вертикальных линий

Рис. 4.17 Результаты применения разных функций интерполяции при отображении кривых

С другой стороны, функции `d3.curveMonotoneX` и `d3.curveCatmullRom` создают кривые, точно следующие по точкам данных и похожие на исходную линейную диаграмму. `d3.curveStep` тоже может создать интересную интерпретацию данных в подходящем контексте. Список на рис. 4.17, перечисляющий функции интерполяции кривых, далеко не полный, а кроме того, некоторые из этих интерполяторов принимают дополнительные параметры, влияющие на форму конечной кривой. Полную информацию о доступных функциях и их параметрах вы найдете в модуле d3-shape.

Теперь вы знаете, как рисовать линейные диаграммы с помощью D3! Давайте коротко вспомним, как это делается: сначала нужно инициализировать функцию-генератор линии и добавить в цепочку к ней функции доступа `x()` и `y()`, как показано на рис. 4.18. Они будут отвечать за расчет горизонтального и вертикального положений каждой точки данных. Затем ломаную линию можно превратить в кривую, добавив в цепочку вызов функции `curve()` и передав ей подходящую интерполяцию. Наконец, элемент `path` добавляется

в диаграмму, и настраивается его атрибут `d` вызовом генератора линий и передачей данных в качестве аргумента. В главе 7 мы сделаем эту линейную диаграмму интерактивной, добавив всплывающие подсказки. Но если хотите узнать об этом прямо сейчас, то смело переходите к главе 7!

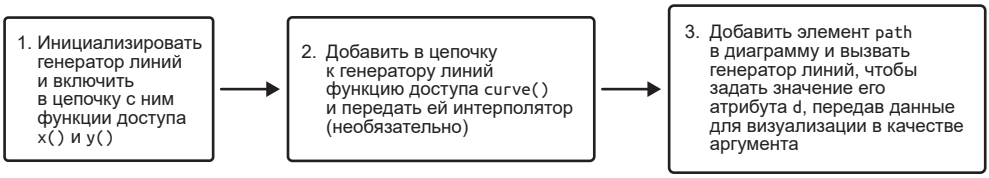


Рис. 4.18 Шаги по созданию линейной диаграммы

4.3 Рисование области размаха

В этом разделе мы добавим область размаха позади линейной диаграммы, показывающую диапазон между минимальной и максимальной температурами для каждой даты. Процесс рисования области в D3 очень похож на процесс рисования линии. Как и линии, области создаются с помощью элементов `path`, и для вычисления атрибута `d` этого элемента D3 предоставляет удобную функцию генератора области, `d3.area()`.

Перед началом давайте уточним еще раз, что мы собираемся отобразить область размаха суточной температуры *за* линейной диаграммой. Поскольку элементы рисуются на экране в том же порядке, в каком они добавляются в родительский элемент SVG, код для рисования области размаха должен находиться в файле JavaScript перед кодом, создающим линейную диаграмму.

4.3.1 Использование генератора области размаха

Сначала объявим функцию-генератор области размаха и сохраним ее в константе с именем `areaGenerator`. Как можно заметить в следующем фрагменте, генератор области требует добавить в цепочку как минимум три функции доступа. Первая, `x()`, отвечает за вычисление горизонтальной координаты точек данных, так же как в случае с генератором линии. Но в этом случае у нас не один набор точек данных, а два: один вдоль нижнего края области и другой верхнего края, это объясняет присутствие функций доступа `y0()` и `y1()`. Обратите внимание, что в нашем случае точки данных на нижнем и верхнем краях области размаха имеют одинаковые горизонтальные координаты:

```
const areaGenerator = d3.area()
  .x(d => xScale(d.date))
  .y0(d => yScale(d.min_temp_F))
  .y1(d => yScale(d.max_temp_F));
```

Рисунок 4.19 помогает понять, как визуализировать нижнюю и верхнюю границы области размаха и как генератор области вычисляет данные, относящиеся к области.

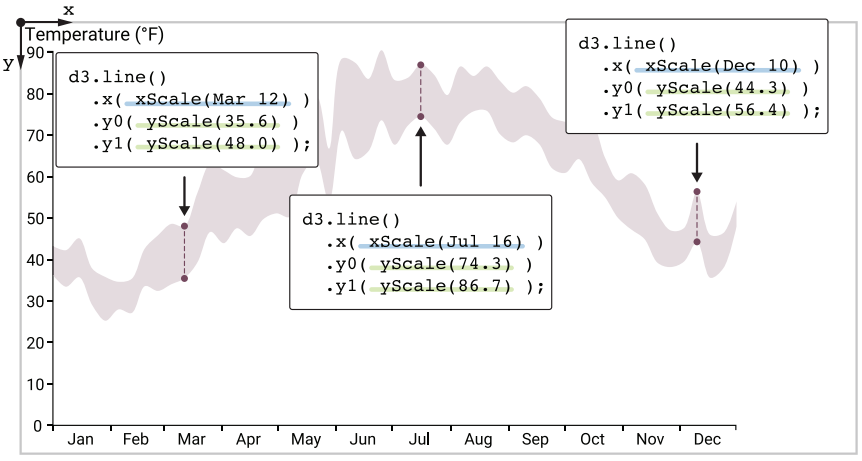


Рис. 4.19 Генератор областей размаха `d3.area()` используется в сочетании с тремя или более функциями доступа. Чтобы нарисовать область между минимальной и максимальной температурами, мы используем функции `x()`, `y0()` и `y1()`. Первая вычисляет горизонтальную координату каждой точки данных, вторая – вертикальные координаты точек данных на нижней границе (минимальная температура), а третья – вертикальные координаты точек данных на верхней границе (максимальная температура)

Так же как в линейной диаграмме, мы интерполируем границы области размаха кривыми, добавив в цепочку функцию `curve()`. Здесь мы используем ту же функцию интерполяции `d3.curveCatmullRom`:

```
const areaGenerator = d3.area()
  .x(d => xScale(d.date))
  .y0(d => yScale(d.min_temp_F))
  .y1(d => yScale(d.max_temp_F))
  .curve(d3.curveCatmullRom);
```

После создания генератора области остается только добавить элемент `path` во внутреннюю диаграмму. Чтобы получить значение для его атрибута `d`, мы вызываем генератор области и передаем набор данных в виде аргумента. Все остальные операции связаны с внешним оформлением. Мы присваиваем атрибуту `fill` константу цвета `aubergine`, объявленную ранее, а атрибуту `fill-opacity` – значение `0.2`, чтобы обеспечить достаточный контраст между областью размаха и линейной диаграммой, как показано на рис. 4.20. Обратите внимание, что константа `aubergine` должна быть объявлена до ее использования:

```
innerChart
  .append("path")
    .attr("d", areaGenerator(data))
    .attr("fill", aubergine)
    .attr("fill-opacity", 0.2);
```

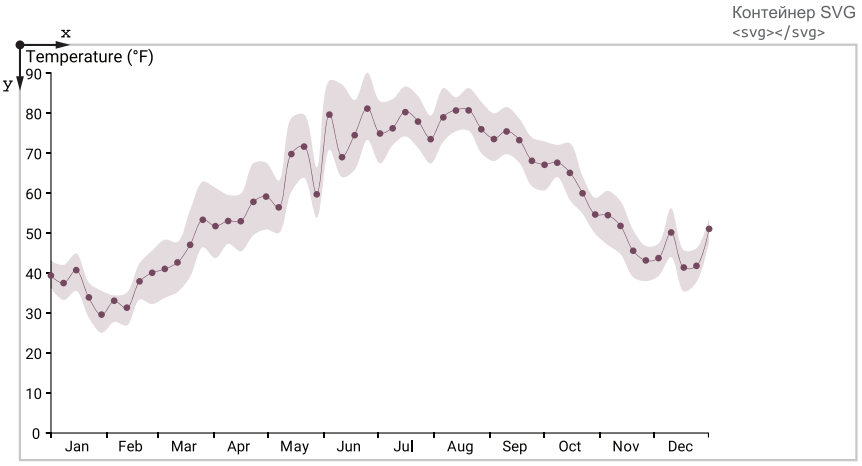


Рис. 4.20 Линейный график средней температуры в сочетании с областью размаха, показывающей разницу между минимальной и максимальной температурами

Как видите, процесс рисования области размаха очень похож на процесс рисования линии. Главное отличие от линии в том, что область размаха – это пространство между двумя линиями, соответствующими двум наборам точек. Вот почему генератору линии требуются только две функции доступа, `x()` и `y()`, а генератору области размаха – как минимум три, в нашем случае `x()`, `y0()` и `y1()` (см. рис. 4.21).

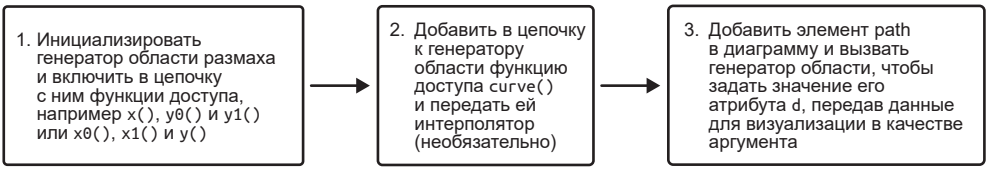


Рис. 4.21 Шаги по созданию области размаха

4.3.2 Улучшение читаемости с помощью меток

Теперь у нас есть линейный график средней температуры в Нью-Йорке за 2021 год, совмещенный с областью размаха, показывающей разницу между минимальной и максимальной температурами. Выглядит неплохо, но нужно также помочь людям, которые увидят этот график, понять, что означают линия и область. Метки – отличный инструмент для этого!

Метки – это простые текстовые элементы SVG, которые можно размещать поверх визуализаций. Здесь мы создадим три метки: одну для описания средней температуры, которую мы разместим в конце линейной диаграммы, одну для описания минимальной температуры, расположенную под областью размаха, и одну для описания максимальной температуры, расположенную над областью размаха.

Начнем с метки для линейной диаграммы. Сначала добавим текстовый элемент SVG во внутреннюю диаграмму и заполним его содержимое строкой «Average temperature» (средняя температура) вызовом метода `text()`. Затем вычислим его местоположение и установим атрибуты `x` и `y`.

Метка должна располагаться в конце линейной диаграммы, сразу после последней точки данных. Мы можем получить эту позицию, используя константу `lastDate`, вычисленную ранее при объявлении шкалы в `xScale()`. Для красоты добавим отступ в10 пикселей.

У нас нет константы со значением температуры в последней точке. Однако мы можем найти последнюю запись в наборе данных с помощью `data[data.length-1]` и использовать точечную нотацию для доступа к средней температуре. Передадим это значение в `yScale()` и получим вертикальную координату метки. Наконец, повторно используем цветовую константу `aubergine`, чтобы присвоить ее атрибуту `fill` и тем самым установить цвет текста:

```
innerChart
  .append("text")
    .text("Average temperature")
    .attr("x", xScale(lastDate) + 10)
    .attr("y", yScale(data[data.length - 1].avg_temp_F))
    .attr("fill", aubergine);
```

Сохраните проект и откройте его в браузере. Вы увидите, что нижняя часть метки выровнена по вертикали с центром последней точки данных на линейной диаграмме. По умолчанию базовая линия текста SVG располагается под текстом, как показано на рис. 4.22. Мы можем изменить ее положение с помощью атрибута `dominant-baseline`. Следующий фрагмент присваивает этому атрибуту значение `middle`, чтобы сместить базовую линию к центру текста по вертикали:

```
innerChart
  .append("text")
    .text("Average temperature")
    .attr("x", xScale(lastDate) + 10)
    .attr("y", yScale(data[data.length - 1].avg_temp_F))
    .attr("dominant-baseline", "middle")
    .attr("fill", aubergine);
```

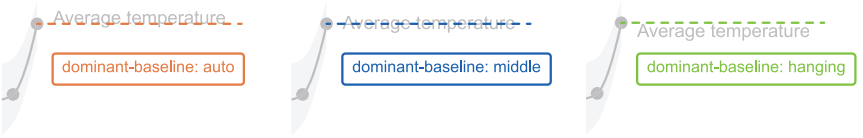


Рис. 4.22 Атрибут у текстового элемента SVG определяет вертикальное положение его базовой линии, которая по умолчанию располагается внизу текста. Мы можем изменить ее положение с помощью атрибута `dominant-baseline`. Если присвоить этому атрибуту значение «middle», то базовая линия текста сместится к центру текста по вертикали, а если присвоить значение «hanging», то она сместится к верхней границе текста

Далее добавим метку для нижней границы области размаха, представляющей эволюцию минимальной температуры. Стратегия очень похожа: сначала добавим текстовый элемент и присвоим ему текст «Minimum temperature» (минимальная температура).

В качестве местоположения выберем последний направленный вниз выступ, соответствующий третьей с конца точке данных. Передадим значения этой точки данных в шкалы, чтобы найти ее координаты, и дополнительно сместим метку вниз на 20 пикселей и вправо на 13 пикселей. Эти числа были подобраны методом проб и ошибок, пока мы не нашли положение, которое выглядело наиболее эстетично. Инспектор браузера здорово помогает при проведении таких небольших экспериментов. Обратите внимание, что мы присвоили атрибуту `dominant-baseline` метки значение `hanging`. Как показано на рис. 4.22, это означает, что атрибут `y` определяет местоположение верхней границы текста.

Наконец, в следующем фрагменте мы добавили вертикальную линию от выступа вниз до метки, чтобы прояснить назначение метки.

На рис. 4.23 показано, как это выглядит. И снова мы использовали шкалы для вычисления атрибутов x1, y1, x2 и y2 линии:

```
innerChart
  .append("text")
    .text("Minimum temperature")
    .attr("x", xScale(data[data.length - 3].date) + 13)
    .attr("y", yScale(data[data.length - 3].min_temp_F) + 20)
    .attr("alignment-baseline", "hanging")
    .attr("fill", aubergine);
innerChart
  .append("line")
    .attr("x1", xScale(data[data.length - 3].date))
    .attr("y1", yScale(data[data.length - 3].min_temp_F) + 3)
    .attr("x2", xScale(data[data.length - 3].date) + 10)
    .attr("y2", yScale(data[data.length - 3].min_temp_F) + 20)
    .attr("stroke", aubergine)
    .attr("stroke-width", 2);
```

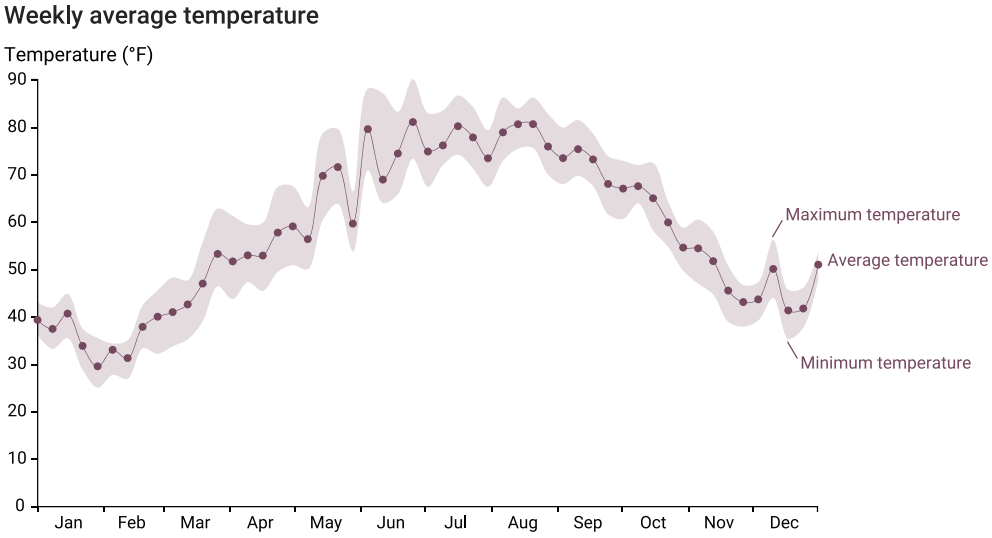


Рис. 4.23 Завершенный линейный график изменения температуры в Нью-Йорке в 2021 году

Аналогично добавляется метка для верхней границы области размаха, представляющей эволюцию максимальной температуры. Эта метка размещается вблизи выступа вверх, соответствующего четвертой с конца точке данных. И снова добавляем линию между меткой и выступом. После добавления следующего фрагмента линейная диаграмма будет завершена:

```
innerChart
  .append("text")
```

```
    .text("Maximum temperature")
    .attr("x", xScale(data[data.length - 4].date) + 13)
    .attr("y", yScale(data[data.length - 4].max_temp_F) - 20)
    .attr("fill", aubergine);
innerChart
  .append("line")
    .attr("x1", xScale(data[data.length - 4].date))
    .attr("y1", yScale(data[data.length - 4].max_temp_F) - 3)
    .attr("x2", xScale(data[data.length - 4].date) + 10)
    .attr("y2", yScale(data[data.length - 4].max_temp_F) - 20)
    .attr("stroke", aubergine)
    .attr("stroke-width", 2);
```

4.4 Рисование дуг

В этом последнем разделе посмотрим, как рисовать дуги с помощью D3. Дуги широко используются в круговых диаграммах, диаграммах «солнечные лучи» и диаграммах «роза Найтингейла» для визуализации соотношения отдельных величин с общей суммой, и мы тоже регулярно используем их.

Подобно линиям и областям размаха, дуги рисуются с помощью SVG-элемента `path`, и, как вы наверняка догадались, D3 предоставляет удобную функцию генератора дуг, которая вычисляет значение атрибута `d` для элемента `path`, изображающего дугу. Прежде чем подробно обсуждать генератор дуг, давайте подготовим демонстрационный проект. В нем мы нарисуем дуги, составляющие радиальную диаграмму, которую можете видеть в разделе «Дни с осадками» на рис. 4.1 или в онлайн-версии проекта (http://mng.bz/5orB). Синяя дуга показывает процент дней с осадками в Нью-Йорке в 2021 году (35 %), а серая дуга – остальные дни.

Сначала откройте файл `arcs.js`. Мы будем работать с ним до конца главы. Как обычно, загрузим набор данных – в данном случае `daily_precipitation.csv`, – который находится в папке `data`. Заглянув в файл CSV, можно увидеть в нем только два столбца: столбец `date` содержит все дни 2021 года, а столбец `total_ precip_in` содержит общее количество осадков в дюймах в каждый день.

В следующем фрагменте мы извлекаем набор данных с помощью `d3.csv()`, используем `d3.autoType` для форматирования дат и чисел и связываем загрузку с объектом `Promise`, в методе `then` которого выводим данные в консоль:

```
d3.csv("./data/daily_precipitation.csv", d3.autoType).then(data => {
  console.log("precipitation data", data);
});
```

> **ПРИМЕЧАНИЕ** Далее мы не будем вдаваться в детали использования `d3.csv()`. Дополнительные объяснения, касающиеся этой функции, вы найдете в главе 3, а описание `d3.autoType` – в разделе 4.1.

Беглого взгляда на данные в консоли достаточно, чтобы увидеть, что даты и числа отформатированы правильно. Отлично! Теперь возьмем отформатированный набор данных и передадим его функции `drawArc()`, которая уже определена в `arcs.js`:

```
d3.csv("./data/daily_precipitation.csv", d3.autoType).then(data => {
  console.log("precipitation data", data);
  drawArc(data);
});
```

Внутри `drawArc()` добавим новый контейнер SVG. Как показано в следующем фрагменте, ширина и высота контейнера устанавливаются равными 300 пикселям, и затем он добавляется внутрь `div` с идентификатором `arc`, уже существующим в `index.html`. Чтобы сделать SVG адаптивным, используется стратегия, описанная в главе 1: два последних значения атрибута `viewBox` устанавливаются равными ширине и высоте, и полностью опускаются атрибуты `width` и `height`. Теперь контейнер SVG будет адаптироваться к размеру родительского элемента, сохраняя заданное соотношение сторон. Обратите внимание, что выборка с контейнером SVG сохраняется в константе с именем `svg`:

```
const pieChartWidth = 300;
const pieChartHeight = 300;
const svg = d3.select("#arc")
  .append("svg")
  .attr("viewBox", [0, 0, pieChartWidth, pieChartHeight]);
```

4.4.1 Полярная система координат

Следуя обсуждению в разделе 4.1, заключим диаграмму в группу SVG и переместим эту группу в желаемое местоположение. Однако на этот раз используем немного иную стратегию. Нам не нужно резервировать место для осей или меток, поэтому мы можем пропустить соглашение о границах и отступах. Но в отличие от других визуализаций, построенных до сих пор, дуги используют полярную систему координат, а не декартову.

Как показано на рис. 4.24, контейнер SVG использует декартову систему координат – две перпендикулярные оси, `x` и `y`, для описания местоположений в двумерном пространстве. Двумерная полярная система координат тоже использует два измерения: радиус и угол. Радиус – это расстояние от начала координат до точки в пространстве, а угол

отсчитывается от 12 часов по часовой стрелке. Такой способ описания положений в пространстве особенно удобен при работе с дугами.

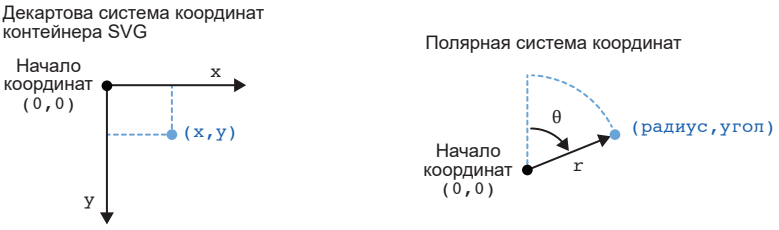


Рис. 4.24 В декартовой системе координат роль измерений играют перпендикулярные оси, а в полярной – радиус и угол

Поскольку элементы позиционируются в полярной системе координат, то можно сказать, что начало системы координат визуализации с дугами, которую мы строим, находится в центре контейнера SVG, как показано на рис. 4.25.

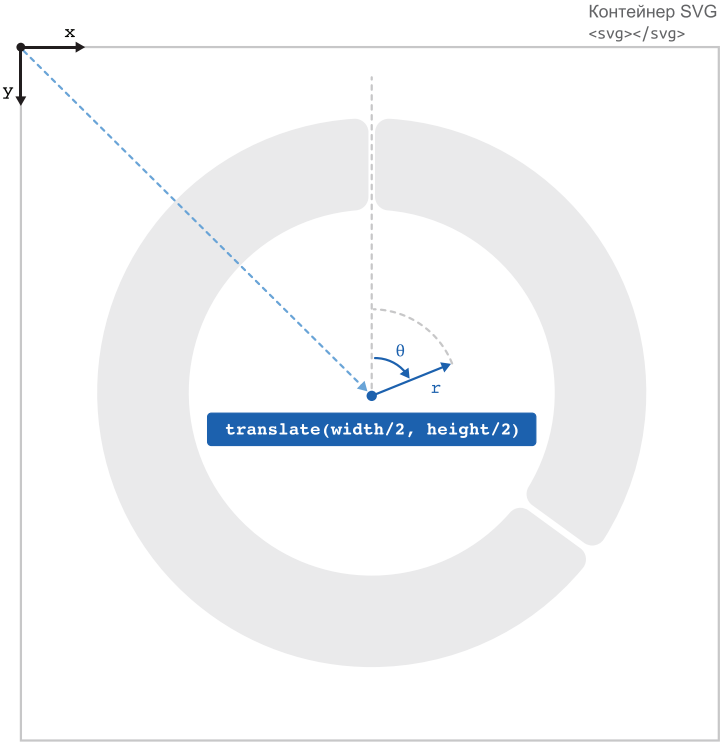


Рис. 4.25 Включение дуг в группу SVG и последующее перемещение группы в центр контейнера SVG упрощает создание набора дуг. При добавлении новых дуг в группу они автоматически будут позиционироваться относительно центра диаграммы – начала полярной системы координат

Следующий фрагмент использует выборку с контейнером SVG, чтобы добавить группу внутрь него. Затем группа перемещается в центр контейнера SVG и сохраняется в константе `innerChart`:

```
const innerChart = svg
  .append("g")
    .attr("transform", `translate(${pieChartWidth/2},
    ➥ ${pieChartHeight/2})`);
```

Прежде чем создавать дуги, сделаем еще кое-что: вычислим угол, который на диаграмме займут дни с осадками. При создании круговых или кольцевых диаграмм с помощью D3 такие вычисления обычно производятся с помощью генератора круговой диаграммы, который мы представим в следующей главе. Но поскольку здесь нам нужно нарисовать всего две дуги, мы просто воспользуемся несложными математическими расчетами.

Сначала узнаем общее количество дней в 2021 году с помощью свойства `length` набора данных (оно равно 365). Затем найдем количество дней с осадками, отфильтровав набор данных и оставив только дни, в которые количество осадков больше 0, что составляет 126 дней. Наконец, мы превратим количество дней с осадками в процент, разделив его на общее количество дней, что дает 35 %:

```
const numberOfDays = data.length;
const numberOfDaysWithPrecipitation = data.filter(d =>
➥ d.total_precip_in > 0).length;
const percentageDaysWithPrecipitation =
➥ Math.round(numberOfDaysWithPrecipitation / numberOfDays * 100);
```

Затем вычислим угол, соответствующий количеству дней с осадками, умножив полученное число на 360° (количество градусов в полной окружности) и получив 126°. Мы начинаем с градусов, потому что это более понятная единица измерения, но далее градусы нужно преобразовать в радианы. Для этого умножим угол 126°, соответствующий количеству дней с осадками, на число пи (3,1416) и разделим на 180. В результате получаем примерно 2,2 радиана. Это значение сохраняем в константе `angleDaysWithPrecipitation_rad`.

Это преобразование необходимо, чтобы воспользоваться генератором дуг, который принимает углы в радианах, а не в градусах. Как правило, при работе с углами в JavaScript обычно используются радианы, а в CSS – градусы:

```
const angleDaysWithPrecipitation_deg = percentageDaysWithPrecipitation *
➥ 360 / 100;
const angleDaysWithPrecipitation_rad = angleDaysWithPrecipitation_deg *
➥ Math.PI / 180;
```

4.4.2 Использование генератора дуг

Наконец-то мы добрались до самой интересной части: генерирования дуг! Сначала объявим генератор дуг, следуя аналогии с линиями и областями размаха. Генератор дуг `d3.arc()` находится в модуле d3-shape и требует двух основных функций доступа: внутреннего и внешнего радиуса дуг, которым соответствуют функции доступа `innerRadius()` и `outerRadius()` со значениями 80 и 120 пикселей. Обратите внимание, что если задать внутренний радиус равным 0, то получится круговая диаграмма:

```
const arcGenerator = d3.arc()
  .innerRadius(80)
  .outerRadius(120);
```

Визуально отделить дуги друг от друга (как показано на рис. 4.26) можно, добавив отступ между ними с помощью функции доступа `padAngle()`, которая принимает угол в радианах. Здесь мы используем отступ, равный 0,02 радиана, т. е. немного больше 1°. Также можно скруглить углы дуг с помощью функции `cornerRadius()`, которая принимает значение в пикселях. Она дает тот же эффект, что и CSS-свойство `border-radius`:

```
const arcGenerator = d3.arc()
  .innerRadius(80)
  .outerRadius(120)
  .padAngle(0.02)
  .cornerRadius(6);
```

Возможно, у вас возник вопрос, почему для обработки углов, охватываемых дугами, мы не использовали функции доступа. В данном случае мы вручную вычислили угол, поэтому для нас проще передать генератору готовые значения. Но в следующей главе мы покажем, что иногда удобнее пользоваться специализированными функциями.

Итак, давайте добавим первую дугу, показывающую количество дней с осадками. В следующем фрагменте мы сначала добавляем элемент `path` в выборку с внутренней диаграммой. Затем устанавливаем его атрибут `d`, вызывая генератор дуг, объявленный в предыдущем фрагменте.

Обратите внимание, что начальный и конечный углы передаются генератору в виде объекта. Значение `startAngle` равно `0`, что соответствует 12 часам, а значение `endAngle` – это угол, соответствующий дням с осадками, рассчитанный ранее. Наконец, мы устанавливаем цвет заливки дуги равным `#6EB7C2` (цианово-синий):

```
innerChart
  .append("path")
```

```
  .attr("d", () => {
    return arcGenerator({
      startAngle: 0,
      endAngle: angleDaysWithPrecipitation_rad
    });
  })
  .attr("fill", "#6EB7C2");
```

```
d3.arc()
  .innerRadius( 80 )
  .outerRadius( 120 )
  .padAngle( 0.02 )
  .cornerRadius( 6 );
```

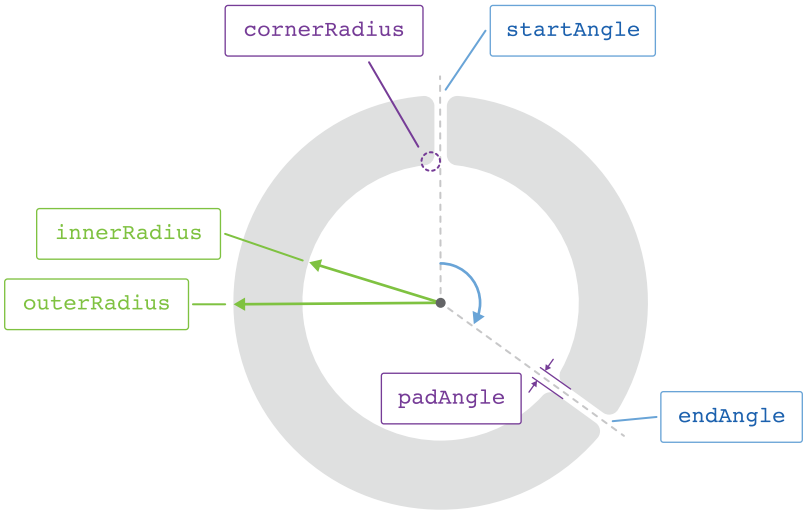


Рис. 4.26 Генератор дуг использует несколько функций доступа для вычисления значения атрибута d. Здесь мы задаем внутренний радиус, внешний радиус, угол отступа и радиус закругления углов. Начальный и конечный углы каждой дуги мы передадим при добавлении элемента path в диаграмму

Аналогично добавляется вторая дуга. На этот раз дуга начинается там, где закончилась предыдущая, и заканчивается в конце окружности, что соответствует углу `2*Pi` в радианах. Для этой дуги устанавливается цвет заливки `#DCE2E2` (ближе к серому), чтобы показать, что это дни без осадков:

```
innerChart
  .append("path")
    .attr("d", () => {
      return arcGenerator({
        startAngle: angleDaysWithPrecipitation_rad,
        endAngle: 2 * Math.PI
```

```
  });
})
.attr("fill", "#DCE2E2");
```

После сохранения проекта дуги должны выглядеть, как показано на рис. 4.27. Поэкспериментируйте со значениями, которые передаются в функции доступа генератора, например с радиусами или радиусами закругления углов, чтобы понять, как они изменяют внешний вид дуг.

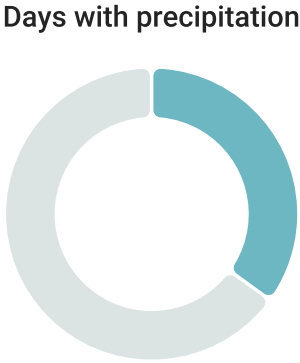


Рис. 4.27 Дуги, показывающие соотношение между днями с осадками и днями без осадков

Как можно видеть на рис. 4.28, процесс рисования дуг похож на процесс рисования линий и областей размаха. Главное отличие – положение дуг в пространстве определяется с использованием полярных координат, а не декартовых, что находит свое отражение в функциях доступа генератора дуг.

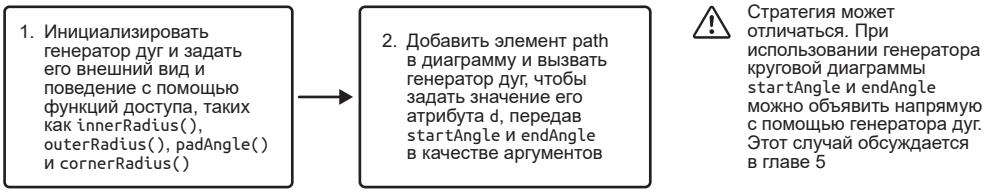


Рис. 4.28 Шаги по созданию дуг

4.4.3 Вычисление центра дуги

Круговые и кольцевые диаграммы в последнее время получили много негативных отзывов в сообществе специалистов по визуализации данных. Главная причина: человеческий глаз плохо оценивает соотношения, представленные углами. Однако иногда эти диаграммы можно считать вполне удачным выбором, особенно когда они содержат небольшое количество категорий. Но мы можем улучшить их читаемость с помощью меток, чем мы и займемся далее.

На дуге, представляющей количество дней с осадками, мы добавим метку с текстом «35 %», описывающую процент дней с осадками, вычисленный ранее. Отличным местом для размещения этой метки является центр дуги, также известный как центроид. Получить координаты центра можно с помощью генератора дуги.

В следующем фрагменте мы вызываем методы генератора дуги, инициализированного ранее. На этот раз мы добавляем в цепочку функции доступа `startAngle()` и `endAngle()`, передавая им соответствующие значения начального и конечного углов дуги, представляющей дни с осадками, и затем метод `centroid()`, вычисляющий координаты середины:

```
const centroid = arcGenerator
  .startAngle(0)
  .endAngle(angleDaysWithPrecipitation_rad)
  .centroid();
```

Если вывести содержимое `centroid` в консоль, то вы увидите, что оно представляет собой массив с двумя значениями: горизонтальной и вертикальной координатами центра относительно начала координат внутренней диаграммы (в нашем случае `[89, -45]`).

В следующем фрагменте мы создаем метку, добавляя текстовый элемент во внутреннюю диаграмму. Чтобы добавить в метку знак «%», мы использовали метод `d3.format(".0%")`, за которым следует значение в скобках. Этот метод удобно использовать для форматирования чисел определенным образом, например валют, процентов и показателей степени, или для добавления определенного суффикса, например «M» для обозначения миллионов или «μ» для обозначения микро. Полный список доступных форматов можно найти в модуле d3-format.

Затем мы устанавливаем атрибуты `x` и `y`, используя первое и второе значения из массива `centroid`. Обратите внимание, что мы устанавливаем атрибуты `text-anchor` и `dominant-baseline`, чтобы гарантировать центрирование метки по атрибутам `x` и `y` как по горизонтали, так и по вертикали.

Наконец, для удобочитаемости придаем метке белый цвет и толщину шрифта 500. Теперь дуга с меткой должна выглядеть, как показано на рис. 4.29:

```
innerChart
  .append("text")
    .text(d => d3.format(".0%")(percentageDaysWith
    ➥ Precipitation/100))
    .attr("x", centroid[0])
    .attr("y", centroid[1])
    .attr("text-anchor", "middle")
    .attr("dominant-baseline", "middle")
```

```
    .attr("fill", "white")
    .style("font-weight", 500);
```

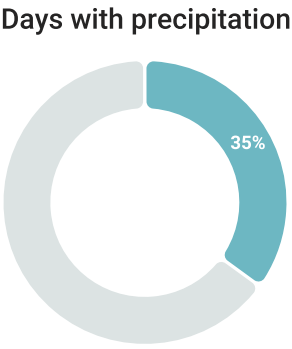


Рис. 4.29 Готовая дуга с меткой

Теперь вы умеете рисовать линии, области размаха и дуги! В следующей главе мы воспользуемся генераторами макетов, чтобы вывести эти фигуры на новый уровень.

Интервью с Фрэнсисом Ганьоном, Патрисией Ангкириванг и Оливией Гелинас из агентства информационного дизайна Voilà

- Ганьон – основатель Voilà и информационный дизайнер.
- Ангкириванг – дизайнер и разработчик визуализаций данных.
- Гелинас – разработчик полного цикла, бывший программист в Voilà.

Расскажите немного о своем агентстве информационного дизайна.

Фрэнсис: Основное внимание мы уделяем проектам устойчивого развития, т. е. всему, что связано с защитой окружающей среды, социальным развитием и хорошим управлением. Мы также считаем себя сторонниками информационного дизайна. Предложения сделать мир лучше могут показаться сложными на первый взгляд, и они конкурируют за внимание с мощными маркетинговыми кампаниями. Мы хотим помочь нашим клиентам донести до окружающих важность своих идей и добиться максимального эффекта.

Какое место занимает D3.js в спектре инструментов визуализации данных, регулярно используемых вашей командой? Какая роль отводится ей в вашем рабочем процессе (от мозгового штурма до создания проектов, готовых к промышленной эксплуатации)?

Патрисия: Наша команда регулярно использует Microsoft Excel и Adobe Illustrator, а также другие инструменты, такие как RAWGraphs, для визуализации статических данных. D3.js дополняет Svelte в нашем рабочем процессе создания интерактивных визуализаций, который мы используем для прототипирования и создания дополнительных визуализаций. Нам нравится эта библиотека за наличие в ней удобных вспомогатель-

ных функций (например, d3-array, d3-format, d3-dsv), а также шкал (d3-scale) и интерполяторов (d3-interpolate), которые мы часто применяем для визуализации наших данных. Также нередко мы используем в своих диаграммах модуль d3-shape.

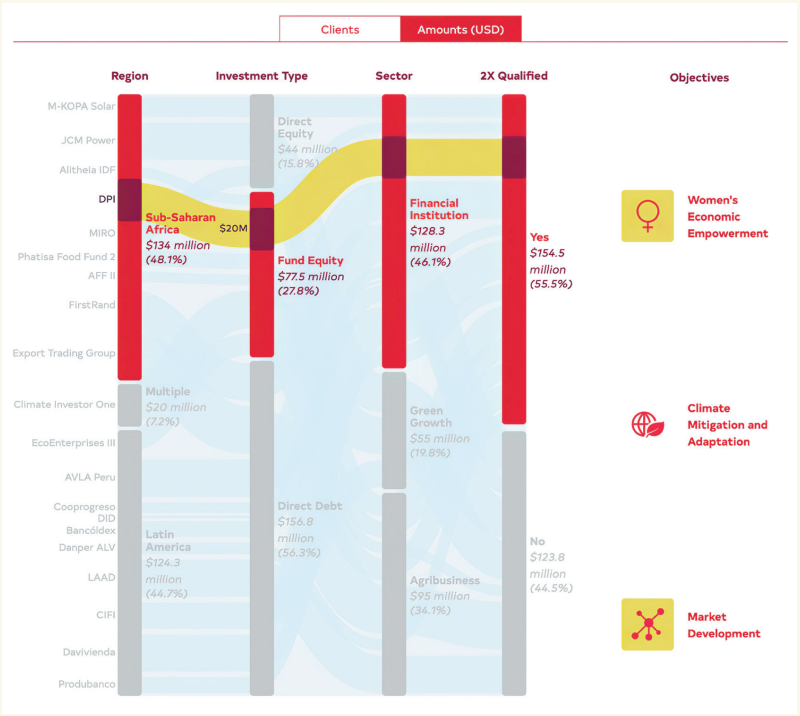
Какие основные достоинства и недостатки D3 вы могли бы назвать, опираясь на свой опыт, особенно на опыт командной работы над проектом?

Патрисия: Я бы сказала, что основное достоинство – способность D3 служить общим языком и наличие подробной документации с примерами. К недостаткам я бы отнесла сложность освоения библиотеки, особенно для нетехнических специалистов в проектах.

Оливия: Я бы добавила, что D3 дает гибкость, недостижимую при использовании других инструментов визуализации. Она упрощает создание практически любых визуализаций, которые вы только сможете придумать, и имеет удобные функции для многих сложных визуализаций (например, диаграммы Сэнки или хорд). Благодаря ее гибкости нетехнические специалисты в команде быстро понимают, что с ее помощью можно реализовать «что угодно», что в их умах быстро превращается во «все подряд»; нетехническим специалистам трудно понять, какие типы задач требуют больше времени на реализацию. Поэтому иногда проблема заключается в управлении требованиями и областью применения, а не в технической стороне дела.

Патрисия: Императивный стиль D3 также может сделать код немного сложнее для чтения. Когда мы создали первую версию этого конкретного проекта, в нем повсюду использовалась D3.js, включая управление элементами SVG и обработку интерактивности. Когда я занялась адаптацией проекта под наш новый рабочий процесс, то выявление всех обработчиков событий, которые использовали модуль d3-select для управления каждым элементом по отдельности, было похоже на собирание запутанной головоломки!

Представленный проект: «A Portfolio of Inclusive Business for a Sustainable Future» (https://chezvoila.com/project/findevportfolio/)



Интерактивный портфель международных проектов развития, визуализированный с помощью аллювиальной диаграммы. Создано для FinDev Canada

Опишите ваш проект, созданный для FinDev Canada.

Фрэнсис: Это интерактивный портфель международных проектов развития. FinDev Canada – организация, которая поддерживает рост и устойчивость бизнеса на развивающихся рынках. Представители организации попросили нас создать что-то, что информирует общественность о характере текущих проектов и одновременно служит панелью управления для их сотрудников. Центральную часть проекта занимает аллювиальный график, позволяющий представлять различные распределения одной и той же переменной. В данном случае такой переменной является финансовая стоимость (или количество) проектов, распределенная по регионам, типам инвестиций, секторам и правоотношениям. Каждый проект становится видимым при наведении указателя мыши на график, и каждый столбец аллювиального графика содержит пропорциональное распределение проектов по категориям.

Опишите кратко, как создавался этот проект.

Фрэнсис: Нашей отправной точкой был предоставленный организацией слайд PowerPoint с данными и графиками, которые на первый взгляд выглядели разрозненными и никак не связанными. Изучив его, мы поняли, что многие из категорий на самом деле представляют одни и те же данные, только по-разному. Поэтому мы решили создать визуализацию, которая проясняла бы эту связь и тем самым передавала природу данных. Наша команда исследовала множество идей для построения такой визуализации, прежде чем остановиться, был выбран аллювиальный график. Затем мы смоделировали дизайн, включая анимацию, в PowerPoint. Первый дизайн, который мы представили, был более отполированным, чем тот, что мы делаем сейчас, что было большим риском для нас, но он оправдался.

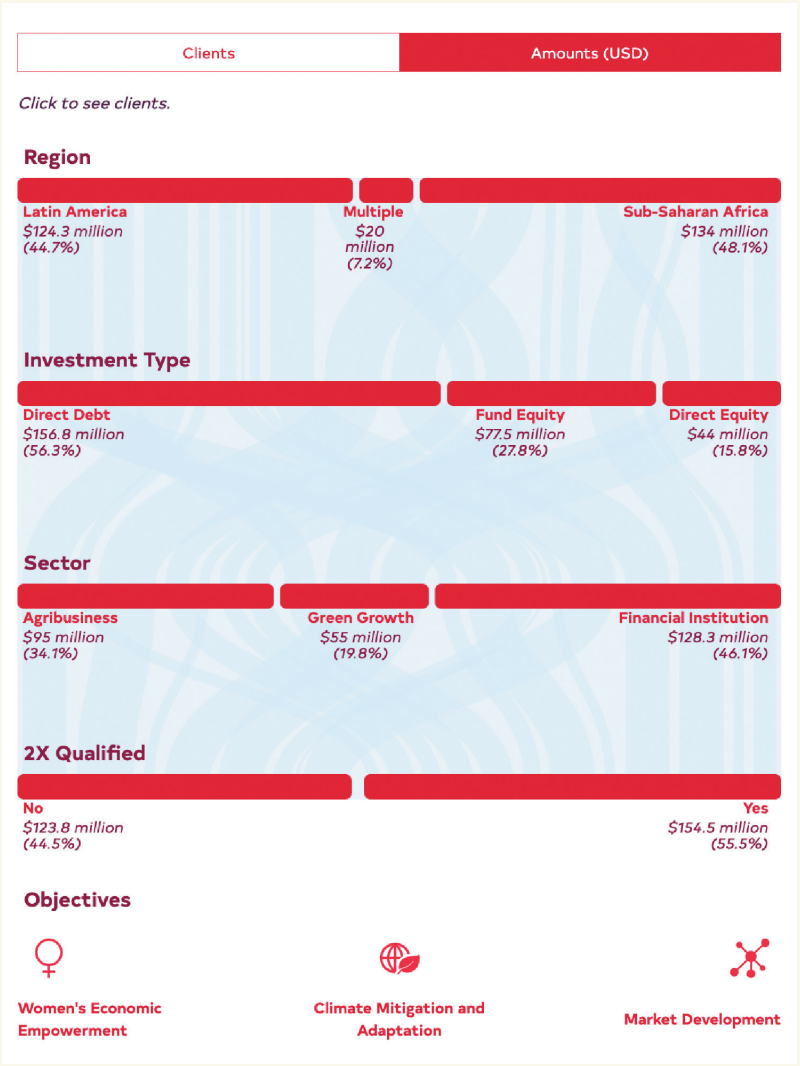
Оливия: После того как макеты были одобрены, осталось лишь реализовать графику. Я составила (на бумаге!) список основных файлов, классов и функций, которые будут использоваться. Затем реализовала код, следуя методологии сверху вниз и начав с определения классов и функций, и закончила всем остальным. Я работала итеративно, время от времени советуясь с командой и внося необходимые коррективы. С тех пор итерации продолжаются, поскольку код периодически дополняется новыми данными и другими улучшениями.

Вы создавали аллювиальный график с помощью модуля d3-sankey или разработали собственное решение?

Патрисия: Да, мы использовали модуль d3-sankey для расчета позиций узлов и связей, а также формы путей связей (с помощью `sankeyLinkHorizontal()`). Для настройки были созданы нестандартные сортировщики, определяющие порядок узлов и связей. После этого нам оставалось реализовать визуальное оформление и интерактивность!

Ваше решение по адаптации визуализации к мобильным экранам простое, но очень эффективное и элегантное. Создание адаптивных визуализаций часто сопряжено с большими сложностями, как эти сложности преодолели вы? Решение было очевидно или вы шли путем проб и ошибок?

Фрэнсис: Нам сопутствовала удача. Уже в конце процесса заказчик заговорил о мобильной версии, к тому же это был наш первый опыт! Мы рассматривали график с минуту, и нас осенило: просто повернуть его на 90°. Первые же попытки дали окончательный результат, и мы сказали заказчику, что так было задумано с самого начала. Потребовалось лишь кое-что настроить, чтобы добавить поддержку касаний к сенсорному экрану.



Поворот диаграммы на 90° позволил получить простое, но элегантное решение для повышения адаптивности

Что бы вы могли посоветовать нашим читателям, изучающим D3 и желающим стать профессиональными разработчиками на D3?

Патрисия: Не стремитесь целиком и полностью «овладеть» D3. Даже я, профессионально используя D3, не знаю и половины того, что может предложить эта библиотека! Я думаю, полезно рассматривать D3 как набор инструментов, которые можно использовать по отдельности. По мере обучения выбирайте интересующие вас модули и используйте их в своих проектах. Лично для меня большим открытием стал модуль d3-scale, изменивший мои подходы к визуализациям.

Итоги

- Соглашение о границах и отступах, принятое в D3, помогает резервировать пространство для осей, меток и легенд систематическим способом.
- С этой целью мы объявляем объект `margin`, содержащий значение для верхнего, правого, нижнего и левого отступов.
- Очень полезно заключать элементы, составляющие саму диаграмму, в группу SVG и размещать ее внутри контейнера SVG с учетом отступов. В результате вы получаете точку отсчета для размещения элементов диаграммы, что упрощает ее реализацию.
- D3 имеет четыре генератора осей: `axisTop()`, `axisRight()`, `axisBottom()` и `axisLeft()`, которые создают соответствующие компоненты верхней, правой, нижней и левой осей.
- Генераторы осей принимают шкалу и возвращают элементы SVG, составляющие ось (линию вдоль оси и несколько наборов рисок и меток).
- Мы добавляем оси, добавляя в цепочку вызов метода `call()` и передавая ему генератор оси.
- Линейные диаграммы – одни из самых распространенных диаграмм. Их удобно использовать для отображения развития явления с течением времени. Мы рисуем линейные диаграммы, используя прямые или кривые, соединяющие точки данных:
  - сначала мы инициализируем генератор линий с помощью метода `d3.line()`. Генератор линий имеет две функции доступа, `x()` и `y()`, которые вычисляют горизонтальное и вертикальное положения каждой точки данных;
  - линейный график можно превратить в кривую с помощью функции доступа `curve()`. D3 предлагает несколько функций интерполяции кривых, влияющих на форму кривой, и их надо выбирать со всем тщанием;
  - чтобы отобразить линейную диаграмму на экране, мы добавляем элемент `path` и настраиваем его атрибут `d`, вызывая генератор линий и передавая набор данных.
- Область размаха – это пространство между двумя границами, и рисование области размаха с помощью D3 похоже на рисование линии:
  - чтобы нарисовать область размаха, мы сначала объявляем генератор области с помощью метода `d3.area()`, который принимает по крайней мере три функции доступа для вычисления координат каждой точки данных вдоль краев области, например `x()`, `y0()` и `y1()` или `x0()`, `x1()` и `y()`;
  - как и для линий, D3 предоставляет функции интерполяции, которые можно применять с помощью функции доступа `curve()`;

  - чтобы область появилась на экране, мы добавляем элемент `path` и настраиваем его атрибут `d`, вызывая генератор области размаха и передавая ему набор данных.
- Метки помогают читателям понять суть визуализации данных. В D3 метки – это просто текстовые элементы, которые нужно разместить в контейнере SVG:
  - местоположение текстового элемента определяется его атрибутами `x` и `y`;
  - атрибут `y` задает местоположение базовой линии текста, которая по умолчанию находится под ним. Мы смещаем базовую линию с помощью атрибута `dominant-baseline`. Значение `middle` перемещает базовую линию к середине текста по вертикали, а значение `hanging` – к верху.
- Визуализации с дугами обычно описываются в полярной системе координат, использующей радиус, расстояние от начала координат до точки и угол.
- Дуги создаются с помощью элементов `path`, атрибут `d` которых настраивается с помощью генератора дуг:
  - генератор дуг D3 (`d3.arc()`) имеет функцию доступа, которая определяет начальный и конечный углы дуги (`startAngle()` и `endAngle()`), а также внутренний и внешний радиусы (`innerRadius()` и `outerRadius()`);
  - также можно использовать функции доступа для скругления углов на концах дуги (`cornerRadius()`) и для добавления отступов между дугами (`padAngle()`);
  - генератор дуги принимает величины углов, выраженные в радианах.
- Центроид дуги можно вычислить с помощью метода `centroid()`. Эта функция, включенная в цепочку вызовов, возвращает массив, содержащий горизонтальное и вертикальное положения центра дуги.

5 Круговые и стековые макеты

В этой главе рассматриваются:

- использование функции макетов D3;
- рисование кольцевых диаграмм с использованием круговых макетов;
- рисование составных столбчатых диаграмм и потоковых графиков;
- создание простых легенд.

В предыдущей главе мы обсудили, как D3 вычисляет атрибут `d` сложных фигур, таких как кривые, области размаха и дуги, используя функции-генераторы. В этой главе мы выведем эти фигуры на новый уровень, применив макеты. Макеты в D3 – это функции, принимающие набор данных и создающие новый, аннотированный набор данных. Этот новый набор данных содержит атрибуты, необходимые для рисования визуализаций определенных видов. Например, круговой макет вычисляет углы каждого фрагмента круговой диаграммы и аннотирует набор данных этими углами. Аналогично стековый макет вычисляет положение фигур в составной столбчатой диаграмме или в потоковом графике.

Макеты не рисуют визуализации и не упоминаются в программном коде, осуществляющем рисование, как генераторы форм, а применя-

ются на этапе предварительной обработки, в ходе которой производится форматирование данных, и подготавливают их к отображению в выбранной форме, как показано на рис. 5.1.

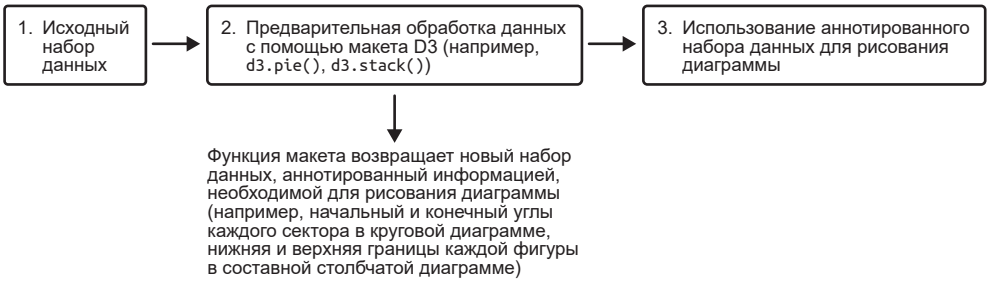


Рис. 5.1 Функции макетов применяются на этапе предварительной обработки данных для расчета информации, необходимой для построения конкретной диаграммы

В этой главе мы объединим круговой и стековый макеты D3 с генераторами дуговых диаграмм и областей размаха, обсуждавшихся в главе 4, и создадим проект, показанный на рис. 5.2. Этот проект можно также найти в интернете по адресу http://mng.bz/6nVo. Он визуализирует продажи по носителям в музыкальной индустрии с 1973 по 2019 год и создан на основе предложения MakeoverMonday, сделанного в 2020 году (www.makeovermonday.co.uk).

В этой главе рассматриваются только круговая и стековая диаграммы, но, следуя тем же принципам, можно создавать и другие диаграммы, такие как аккордовая и более экзотические.

Прежде чем начать, перейдите к файлам кода для главы 5. Их можно загрузить из репозитория книги на GitHub, если вы этого еще не сделали (http://mng.bz/Xqjv). В папке `chapter_05` файлы с кодом организованы по разделам. Чтобы начать выполнение упражнений этой главы, откройте папку `5.1-Pie_layout/start` в редакторе и запустите локальный веб-сервер. Если вам нужна помощь в настройке локальной среды разработки, то обратитесь к приложению A. Более подробную информацию о структуре папок проекта вы найдете в файле `README`.

В этой главе мы построим три визуализации – кольцевые диаграммы, составную столбчатую диаграмму и потоковый график. В них будут использоваться одни и те же данные, измерения и шкалы. Чтобы избежать повторений, проект разбит на несколько файлов JavaScript: один для констант, один для визуализаций и один для шкал. Такая организация упростит чтение и изменение кода. В промышленном коде мы, скорее всего, использовали бы инструкции импортирования и экспортирования для доступа к различным функциям в сочетании с платформой Node. Но об этом мы поговорим в главе 8, когда будем обсуждать фреймворки для создания пользовательских

интерфейсов, а пока будем придерживаться устаревшей структуры проекта, чтобы сосредоточиться на D3. Обратите внимание, что библиотека D3 и все файлы JavaScript уже подключаются в `index.html`.

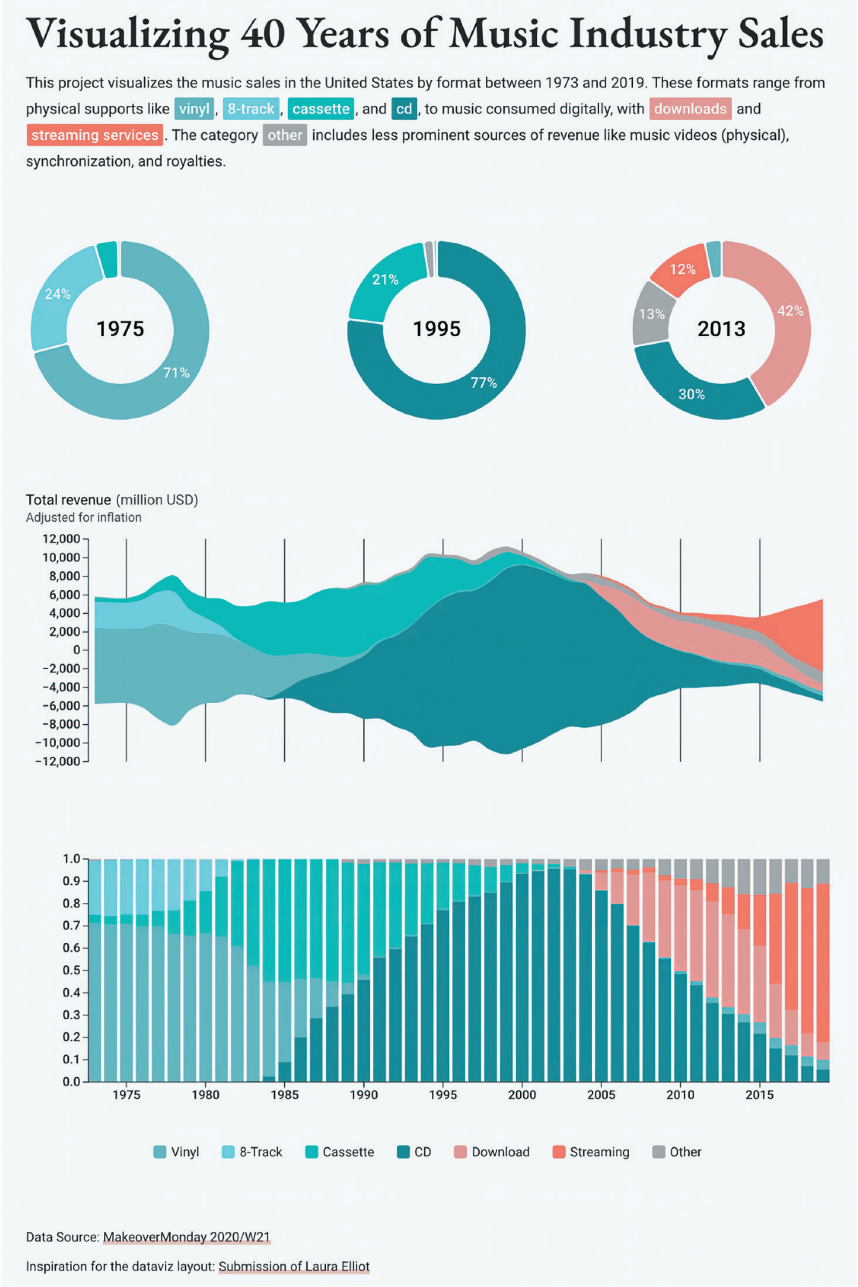


Рис. 5.2 Визуализация продаж в музыкальной индустрии в период с 1973 по 2019 год. Мы построим этот проект в данной главе

ВНИМАНИЕ При работе с примерами кода для этой главы открывайте в редакторе только одну папку, `start` или `end`. Если открыть сразу обе и использовать расширение Live Server для обслуживания проекта, то некоторые пути не будут работать так, как ожидалось.

5.1 Создание круговых и кольцевых диаграмм

В этом разделе мы воспользуемся круговой диаграммой D3 и с ее помощью создадим кольцевые диаграммы, которые можно увидеть в верхней части на рис. 5.2 и в онлайн-версии проекта (http://mng.bz/6nVo). В частности, мы визуализируем распределение продаж по форматам носителей за 1975, 1995 и 2013 годы. Центр каждой кольцевой диаграммы будет соответствовать положению соответствующих лет на оси `x` потокового графика и стековой диаграммы ниже.

5.1.1 Подготовительные шаги

Давайте уделим немного времени стратегии, которая обеспечит правильное горизонтальное выравнивание каждой диаграммы в соответствии с годами на оси `x`. Самый простой способ – использовать соглашение о границах и отступах, описанное в главе 4. В этой главе мы используем три контейнера SVG: один для кольцевых диаграмм, один для потокового графика и один для стековой диаграммы. Все контейнеры будут иметь одинаковые размеры и отступы. Области, зарезервированные для внутренних диаграмм (визуализации без осей и меток), тоже будут иметь одинаковые размеры и выравниваться по горизонтали, как показано на рис. 5.3. Файл `js/shared-constant.js` уже содержит объект `margin` и константы, определяющие размеры визуализаций.

Также в `js/load-data.js` мы загрузили файл данных в формате CSV. О том, как загружать данные в проект D3, рассказывалось в главах 3 и 4. После загрузки данных мы вызываем функции `defineScales()` и `drawDonutCharts()`, которые будем использовать в этом разделе.

Сначала добавим контейнер SVG для кольцевых диаграмм и группу SVG, определяющую область, зарезервированную для внутренней диаграммы. Для этого перейдем в `js/donut-charts.js` и внутри функции `drawDonutCharts()` создадим контейнер и группу SVG. В следующем фрагменте видно, что контейнер SVG добавляется внутрь `div` с идентификатором `"donut"`. Обратите внимание, что мы применяем соглашение о границах и отступах, смещая группу с учетом левого и верхнего отступов диаграмм:

```
const svg = d3.select("#donut")
  .append("svg")
    .attr("viewBox", `0 0 ${width} ${height}`);
```

Добавить контейнер SVG и установить его атрибут viewBox

```
const donutContainers = svg
  .append("g")
    .attr("transform", `translate(${margin.left},
      ${margin.top})`);
```

Добавить группу SVG и сместить ее с учетом левого и верхнего отступов диаграмм

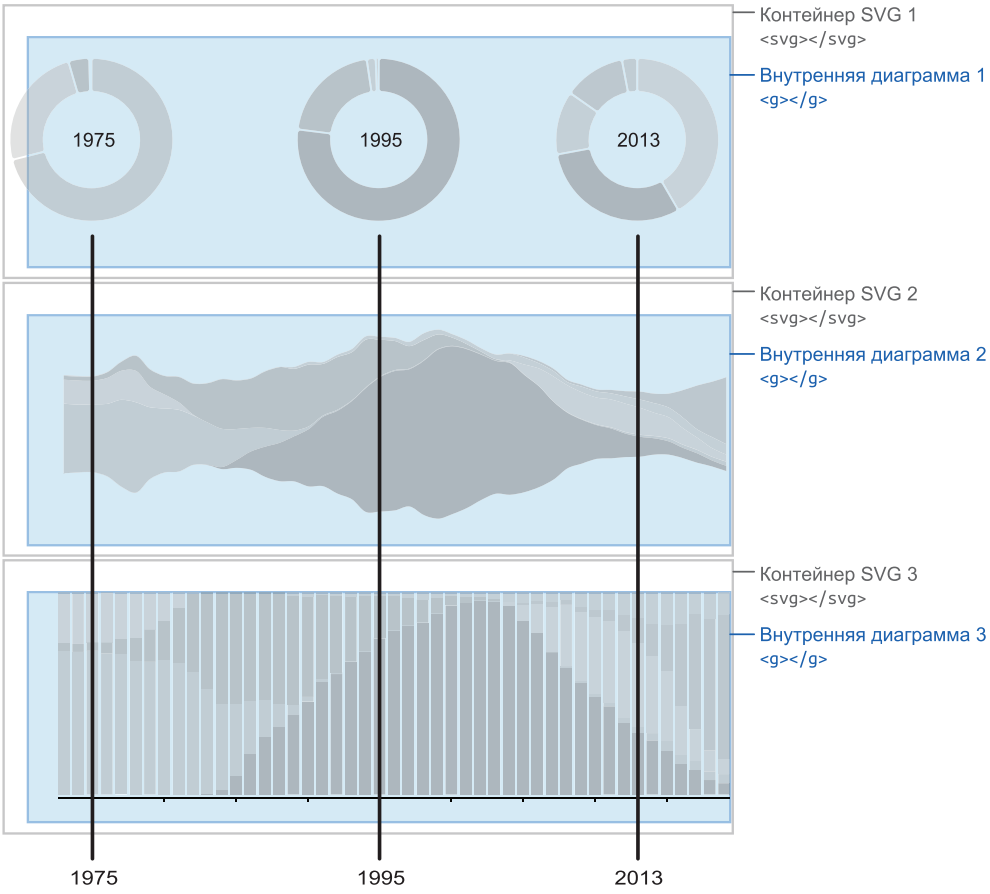


Рис. 5.3 В нашем проекте мы используем три контейнера SVG: один для кольцевых диаграмм, один для потокового графика и один для стековой диаграммы. Такая организация позволит нам согласовать области, зарезервированные для внутренних диаграмм, и правильно выровнять диаграммы по горизонтали

У вас может возникнуть вопрос: зачем применять соглашение о границах и отступах к кольцевым диаграммам, если в них нет ни осей, ни меток, для размещения которых нужно резервировать место? Причина в том, что каждая кольцевая диаграмма будет выровнена с соответствующим годом на оси x. Нам нужно, чтобы эти

диаграммы располагались по горизонтали в соответствии с теми же годами, отображаемыми в потоковом графике и стековой диаграмме ниже, а следовательно, мы должны учесть соглашение о границах и отступах, применяемое в этих диаграммах.

В главе 4 мы познакомились с полярными координатами и узнали, как упростить создание круговой или кольцевой диаграммы, поместив дуги в группу SVG и сместив эту группу в центр диаграммы. При таком подходе дуги автоматически рисуются вокруг этого центра.

Здесь мы применим ту же стратегию, с той лишь разницей, что кольцевых диаграмм будет три, а горизонтальное положение их центров – соответствовать годам, которые они представляют (рис. 5.4).

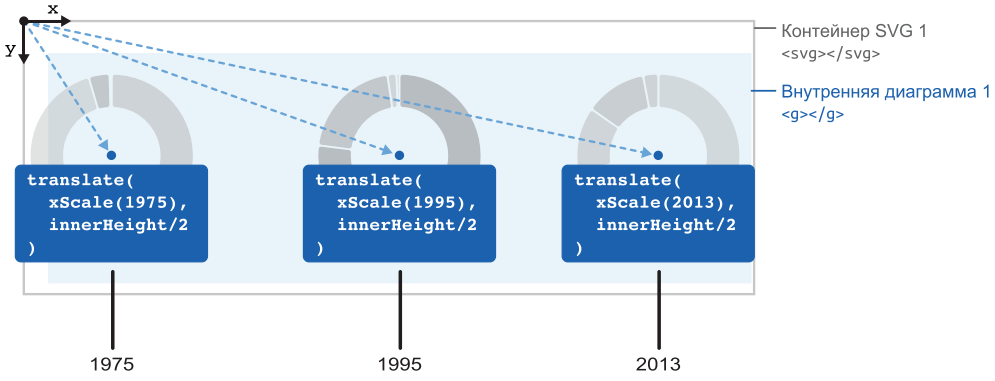


Рис. 5.4 Каждый набор дуг, составляющих кольцевую диаграмму, находится внутри группы SVG. Группы смещаются по горизонтали в зависимости от года, который они представляют. Величина смещения вычисляется с помощью шкал D3

Для вычисления горизонтальной координаты центра каждой кольцевой диаграммы нам понадобится шкала. Как вы уже знаете, шкалы D3 используются для преобразования данных (годов) в экранные атрибуты (горизонтальные координаты). Для наших целей прекрасно подойдет линейная или временная шкала, но воспользуемся интервальной шкалой, зная, что позднее нарисуем составную столбчатую, которая будет иметь ту же шкалу. За дополнительной информацией о работе шкал обращайтесь к главе 3.

В файле `js/scale.js` мы сначала инициализируем интервальную шкалу с помощью функции `d3.scaleBand()` и сохраним ее в константе с именем `xScale`. Обратите внимание, как объявляются предметная область и диапазон шкалы внутри функции `defineScales()`. Такой подход позволяет дождаться завершения загрузки данных, прежде чем задать предметную область. Функция `defineScales()` вызывается из `load-data.js` сразу после загрузки данных. Мы объявляем константу `xScale` вне функции, чтобы сделать ее доступной из других файлов JavaScript.

Листинг 5.1 Объявление интервальной шкалы (scales.js)

```
const xScale = d3.scaleBand();

const defineScales = (data) => {
  xScale
    .domain(data.map(d => d.year))
    .range([0, innerWidth]);
};
```

Объявление интервальной шкалы

Настройка предметной области и диапазона шкалы

Интервальные шкалы принимают дискретную предметную область и возвращают непрерывный диапазон. В листинге 5.1 мы задаем предметную область, создавая массив, содержащий года из набора данных, вызывая для этого JavaScript-метод `map()`. В качестве диапазона передается массив, содержащий минимальную доступную горизонтальную координату, равную 0, и максимальную, соответствующую ширине внутренней диаграммы.

Вернемся к функции `drawDonutCharts()`. В ней, как показано в листинге 5.2, сначала объявляем массив с именем `years`, перечисляющий интересующие нас годы: 1975, 1995 и 2013. Затем, используя цикл `forEach()`, добавляем группы SVG для каждого года и сохраняем их в константе с именем `donutContainer`. И наконец, смещаем группы, используя их атрибуты `transform`. Смещение по горизонтали вычисляется вызовом шкалы `xScale`, которой передается текущий год, а смещение по вертикали соответствует половине высоты внутренней диаграммы.

Листинг 5.2 Добавление группы для каждой кольцевой диаграммы (donut-charts.js)

```
const years = [1975, 1995, 2013];
years.forEach(year => {

  const donutContainer = donutContainers
    .append("g")
      .attr("transform", `translate(${xScale(year)}, ${innerHeight/2})`);

});
```

5.1.2 Генератор кругового макета

Завершив подготовительные шаги, сосредоточимся на кольцевых диаграммах. Круговые и кольцевые диаграммы визуализируют отношения «часть–целое» – количество, представленное каждым сектором, по отношению к общему количеству. Генератор круговых макетов в D3 вычисляет начальный и конечный углы каждого сектора на основе процента, который он представляет.

Форматирование данных

Генератор круговых макетов в D3 принимает входные данные, отформатированные как массив чисел. Например, для 1975 года у нас может быть массив с продажами, соответствующими каждому носителю:

```
const sales1975 = [8061.8, 2770.4, 469.5, 0, 0, 0, 48.5];
```

Такого простого массива достаточно для создания круговой диаграммы, но он не позволяет узнать тип носителя музыки, связанный с каждым сектором. Для передачи этой информации можно использовать массив объектов, содержащий как идентификатор типа носителя, так и объем продаж этого носителя за интересующий нас год.

В листинге 5.3 мы сначала извлекаем типы носителей из атрибута `columns` в наборе данных. Когда данные извлекаются с помощью метода `d3.csv()`, D3 присоединяет к результирующему набору данных массив с заголовками столбцов из исходного набора данных. Он доступен как атрибут `data.columns`. Если вы попробуете вывести извлеченные данные в консоль, то увидите эти заголовки в конце набора данных, как показано на рис. 5.5. Поскольку нас интересуют только типы носителей, мы можем отфильтровать массив `columns`, удалив метку «year».

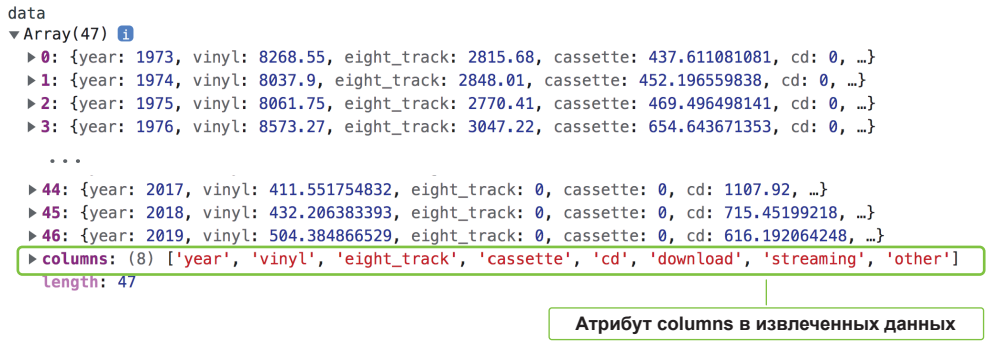


Рис. 5.5 Когда данные извлекаются, например, из CSV-файла, D3 присоединяет к результирующему набору данных массив с заголовками столбцов из исходного набора данных. Он доступен как атрибут `data.columns`

Чтобы подготовить данные для генератора круговой диаграммы, также нужно извлечь данные для интересующего года. Мы выбираем эти данные с помощью JavaScript-метода `find()` и сохраняем их в константе с именем `yearData`.

Далее выполняем обход типов носителей музыки и для каждого создаем объект, содержащий идентификатор типа и связанные

с ним продажи за интересующий год. В заключение мы помещаем этот объект в массив `formattedData`, объявленный ранее.

Листинг 5.3 Форматирование данных (donut-charts.js)

```
const years = [1975, 1995, 2013];
const formats = data.columns.filter(format =>
  format !== "year");

years.forEach(year => {
    ...
    const yearData = data.find(d => d.year === year);

    const formattedData = [];

    formats.forEach(format => {
        formattedData.push({ format: format,
            sales: yearData[format] });
    });
});
```

Извлечь типы из атрибута columns и отфильтровать элемент "year"

Извлечь данные, относящиеся к интересующему году

Инициализировать пустой массив для форматированных данных

Для каждого типа создать объект, содержащий идентификатор типа носителя и объем его продаж за интересующий год. Поместить этот объект в массив отформатированных данных

Отформатированные данные имеют вид массива объектов, каждый из которых содержит идентификатор типа носителя и объем связанных с ним продаж за интересующий год:

```
// => formattedData = [
        { format: "vinyl", sales: 8061.8 },
        { format: "eight_track", sales: 2770.4 },
        { format: "cassette", sales: 469.5 },
        { format: "cd", sales: 0 },
        { format: "download", sales: 0 },
        { format: "streaming", sales: 0 },
        { format: "other", sales: 48.5 }
      ];
```

Инициализация и вызов генератора кругового макета

Теперь, отформатировав данные должным образом, мы можем инициализировать генератор кругового макета. Мы создадим новый генератор с помощью метода `d3.pie()` из модуля d3-shape. Поскольку отформатированные данные имеют вид массива объектов, нам нужно сообщить генератору, какой ключ содержит значение, определяющее размер сектора. Для этого настроим функцию доступа `value()`, как показано в следующем фрагменте, и сохраним генератор в конс-

танте с именем `pieGenerator`, чтобы иметь возможность вызывать его как любую другую функцию:

```
const pieGenerator = d3.pie()
  .value(d => d.sales);
```

Чтобы создать аннотированные данные для кругового макета, просто вызовем функцию генератора, передадим ей отформатированные данные и сохраним результат в константе с именем `annotatedData`:

```
const pieGenerator = d3.pie()
  .value(d => d.sales);
const annotatedData = pieGenerator(formattedData);
```

Генератор кругового макета диаграмм возвращает новый аннотированный набор данных, содержащий ссылку на исходный набор и включающий новые атрибуты – значение для каждого сектора, его индекс, а также его начальный и конечный углы (в радианах):

```
// => annotatedData = [
        {
          data: { format: "vinyl", sales: 8061.8 },
          value: 8061.8,
          index: 0,
          startAngle: 0,
          endAngle: 4.5,
          padAngle: 0,
        },
        ...
      ];
```

Обратите внимание, что в данных также присутствует атрибут `padAngle`, определяющий отступ между секторами, но в настоящий момент он имеет значение 0. Мы изменим его чуть ниже. Важно понимать, что генератор кругового макета не участвует напрямую в рисовании диаграммы. Он относится к этапу предварительной обработки и просто вычисляет углы секторов круговой диаграммы. Как описано на рис. 5.1 и 5.6, этот процесс обычно включает три шага:

1. Форматирование данных.
2. Инициализация функции круговой диаграммы.
3. Вызов макета круговой диаграммы и передача ему отформатированных данных. Позже мы используем аннотированный набор данных, возвращаемый макетом круговой диаграммы, для рисования дуг.

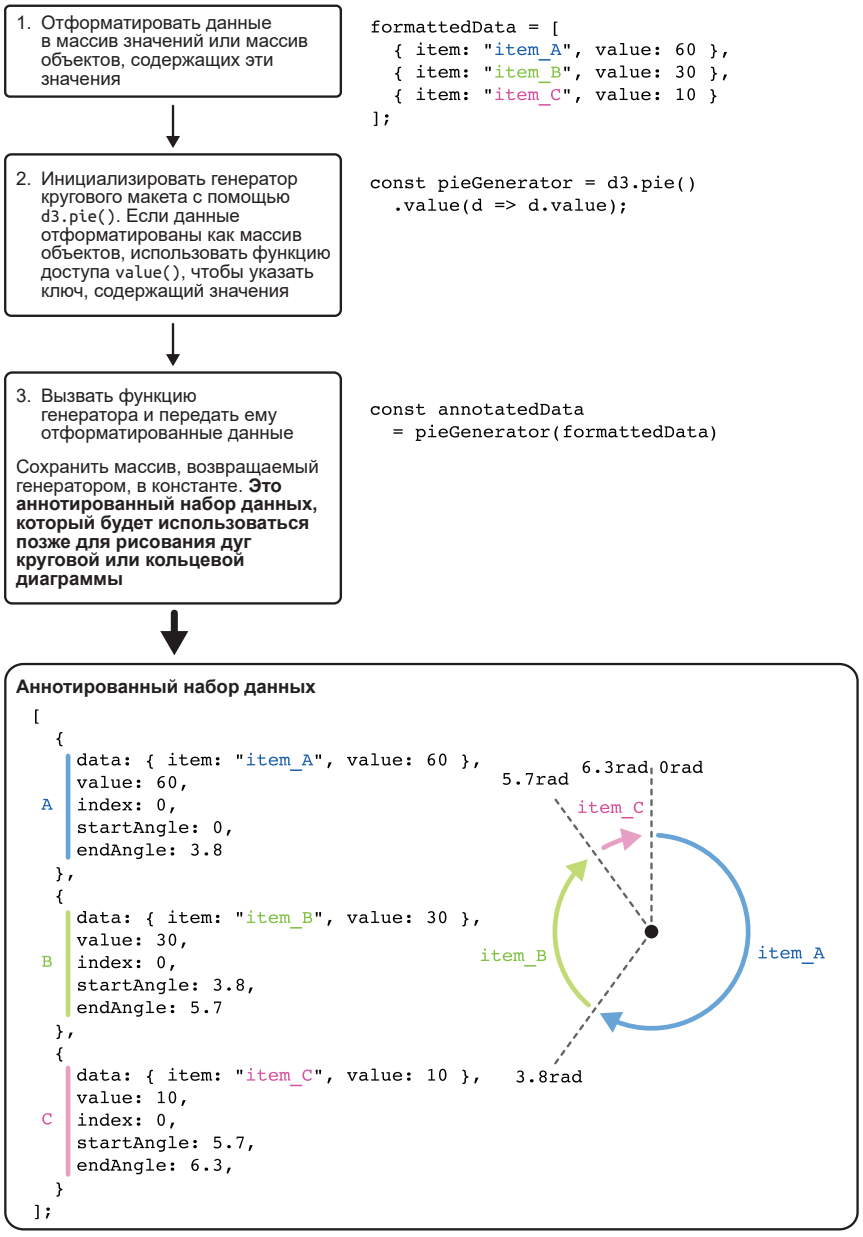


Рис. 5.6 Генератор кругового макета относится к этапу предварительной обработки и просто вычисляет углы секторов круговой диаграммы. Процесс обычно включает форматирование наших данных, инициализацию функции генератора и вызов этой функции для получения аннотированных данных

5.1.3 Рисование дуг

После подготовки аннотированного набора данных можно начинать рисовать дуги круговых диаграмм! Далее вы увидите, что выполняемые при этом шаги очень похожи на рисование дуг в предыдущей главе. Поэтому мы не будем объяснять каждую деталь. Если вам понадобится более глубокое обсуждение, то обратитесь к главе 4.

В листинге 5.4 мы сначала инициализируем генератор дуг, вызывая метод `d3.arc()` и его различные функции доступа, отвечающие за настройку внутреннего и внешнего радиусов диаграммы, отступа между секторами и радиуса углов на концах секторов. Если установить внутренний радиус равным 0, то мы получим круговую диаграмму, а если дать ему значение больше 0, то получим кольцевую диаграмму.

Единственное отличие от стратегии, использованной в главе 4, заключается в том, что на этот раз мы можем задать функции доступа `startAngle()` и `endAngle()` при объявлении генератора дуги, потому что эти значения теперь включены в аннотированный набор данных, и мы можем указать, что функции могут получить к ним доступ через `d.startAngle` и `d.endAngle`.

Последнее, что мы должны сделать для отображения дуг на экране, – использовать шаблон привязки данных, чтобы сгенерировать по одному элементу `path` для каждого объекта в аннотированном наборе данных (каждой дуге или сектору соответствует один объект). Обратите внимание, что в листинге 5.4 мы задаем для дуг конкретное имя класса (`` `arc-${year}` ``) и используем его в качестве селектора в шаблоне привязки данных. Поскольку мы создаем кольцевые диаграммы в цикле, это не позволит D3 затереть предыдущую диаграмму при генерировании новой. Наконец, мы вызываем функцию генератора дуг для вычисления атрибута `d` каждого элемента `path`.

Листинг 5.4 Генерирование и рисование дуг (donut-charts.js)

```
const arcGenerator = d3.arc()
  .startAngle(d => d.startAngle)
  .endAngle(d => d.endAngle)
  .innerRadius(60)
  .outerRadius(100)
  .padAngle(0.02)
  .cornerRadius(3);

const arcs = donutContainer
  .selectAll(`.arc-${year}`)
  .data(annotatedData)
  .join("path")
    .attr("class", `arc-${year}`)
    .attr("d", arcGenerator);
```

Вызвать функции доступа startAngle() и endAngle() и указать им ключ, под которым значения для них доступны в аннотированном наборе данных

Использовать шаблон привязки данных для генерации одного элемента path для каждого объекта в аннотированном наборе данных

Вызвать функцию arcGenerator(), чтобы получить атрибут d для каждого элемента path

Использование цветовой шкалы

Сохранив проект и открыв его в браузере, вы увидите, что кольцевые диаграммы имеют правильную форму, но дуги черные как смоль. Это нормально, потому что черный цвет заливки используется по умолчанию. Для улучшения читаемости применим к каждой свой цвет в зависимости от типа носителя музыки, который она представляет.

Простой способ применить правильный цвет к каждой дуге – объявить цветовую шкалу. В D3 цветовые шкалы часто создаются с помощью `d3.scaleOrdinal()`. Порядковые шкалы отображают дискретную предметную область в дискретный диапазон. В нашем случае предметная область – это массив типов носителей, а диапазон – это массив цветов, соответствующих типам.

В файле `scales.js` мы сначала объявляем порядковую шкалу и сохраняем ее в константе `colorScale`. Затем задаем предметную область, отображая каждый идентификатор типа носителя из массива `formatsInfo` (доступного в `shared-constants.js`) в массив. То же самое мы делаем с цветами, которые вы можете подобрать по своему вкусу. На протяжении всей главы мы не раз будем использовать эту цветовую шкалу для создания всех диаграмм, составляющих наш проект:

```
const colorScale = d3.scaleOrdinal();

const defineScales = (data) => {

  colorScale
    .domain(formatsInfo.map(f => f.id))
    .range(formatsInfo.map(f => f.color));

};
```

Возвращаемся в `donut-charts.js`, где зададим атрибут `fill` дуг, передав в цветовую шкалу идентификатор типа носителя музыки, привязанного к каждой дуге:

```
const arcs = donutContainer
  .selectAll(`.arc-${year}`)
  .data(annotatedData)
  .join("path")
    .attr("class", `arc-${year}`)
    .attr("d", arcGenerator)
    .attr("fill", d => colorScale(d.data.format));
```

Сохраните свой проект и откройте его в браузере – выглядит неплохо! Как показано на рис. 5.7, дуги уже отображаются в порядке убывания, от самой большой к самой маленькой, что улучшает удобочитаемость. Мы уже видим, как менялся облик музыки между

1975, 1995 и 2013 годами, о чем свидетельствуют совершенно разные доминирующие типы носителей.

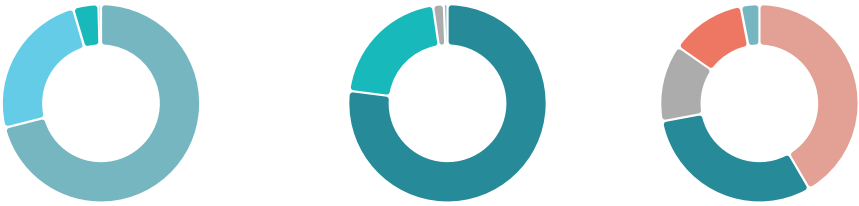


Рис. 5.7 Кольцевые диаграммы за 1975, 1995 и 2013 годы

5.1.4 Добавление меток

В главе упоминалось, что круговые диаграммы иногда трудно интерпретировать, учитывая, что человеческий мозг плохо переводит углы в соотношения. Мы можем улучшить читаемость наших кольцевых диаграмм, добавив метки с процентами, соответствующими секторам, как мы делали это в предыдущей главе.

В листинге 5.5 представлен немного измененный код создания дуг (из листинга 5.4). Во-первых, здесь мы используем шаблон привязки данных для добавления групп SVG. Затем добавляем элементы `path` (для дуг) и текстовые элементы (метки). Поскольку данные, привязанные к родительским элементам, доступны потомкам, мы используем данные по мере формирования дуг и меток.

Для рисования дуг вызывается генератор дуг, как мы делали это ранее. Чтобы задать текст метки, нужно вычислить отношение или процент, представленный каждой дугой. Для этого начальный угол дуги вычитается из ее конечного угла и результат делится на 2π, величину угла в радианах, охватываемого полной окружностью. Обратите внимание, что процентное значение сохраняется в связанных данных с использованием квадратных скобок (`d["percentage"]`). Этот прием очень удобен, когда нужно произвести одно и то же вычисление для разных атрибутов. Он позволяет не повторять вычисление несколько раз. Чтобы вернуть текст метки, мы передаем вычисленный процент методу `d3.format(".0%")`, возвращающему округленное число и добавляющему символ процента в конец метки.

Та же стратегия применяется для вычисления центра (или центроида) каждой дуги, где должна располагаться метка. При установке атрибута `x` меток мы вычисляем центр соответствующей дуги (с помощью приема, обсуждавшегося в главе 4) и сохраняем его в связанных данных (`d["centroid"]`). При установке атрибута `y` массив центроидов будет доступен через `d.centroid`.

Чтобы отцентрировать метки по горизонтали и вертикали, нужно присвоить их атрибутам `text-anchor` и `dominant-baseline` значе-

ние "middle". Мы также присваиваем их атрибутам fill белый цвет, а еще увеличиваем размер шрифта до 16 пикселей и вес шрифта до 500, чтобы улучшить читаемость.

Сохранив и открыв проект в браузере, вы увидите, что метки хорошо различимы на больших дугах, но почти нечитаемы на маленьких. В профессиональном проекте мы бы решили эту проблему, переместив метки для маленьких дуг за пределы кольцевой диаграммы. В данном проекте мы просто не будем показывать эти метки, присвоив их атрибуту fill-opacity значение 0, если процент окажется меньше 5 %. Окончательный результат показан на рис. 5.8.

Листинг 5.5 Добавление меток к дугам (donut-charts.js)

```
const arcs = donutContainer
  .selectAll(`.arc-${year}`)
  .data(annotatedData)
  .join("g")
    .attr("class", `arc-${year}`);

arcs
  .append("path")
    .attr("d", arcGenerator)
    .attr("fill", d => colorScale(d.data.format));

arcs
  .append("text")
    .text(d => {
      d["percentage"] = (d.endAngle - d.startAngle)
        / (2 * Math.PI);
      return d3.format(".0%")(d.percentage);
    })
    .attr("x", d => {
      d["centroid"] = arcGenerator
        .startAngle(d.startAngle)
        .endAngle(d.endAngle)
        .centroid();
      return d.centroid[0];
    })
    .attr("y", d => d.centroid[1])
    .attr("text-anchor", "middle")
    .attr("dominant-baseline", "middle")
    .attr("fill", "#f6fafc")
    .attr("fill-opacity", d => d.percentage
      < 0.05 ? 0 : 1)
    .style("font-size", "16px")
    .style("font-weight", 500);
```

Использовать шаблон привязки данных для добавления групп SVG

Добавить элемент path внутрь каждой группы и нарисовать дуги, вызвав генератор дуг. Связать атрибут fill с цветовой шкалой

Добавить текстовый элемент внутрь каждой группы

Задать текст, вычислив процент, занимаемый каждой дугой. Сохранить полученное значение в связанных данных (d["percentage"])

Получить координаты центроида каждой дуги и сохранить его в связанных данных (d["centroid"]). Затем использовать его для установки атрибутов x и y меток

Скрыть метки для дуг, представляющих значение меньше 5 %

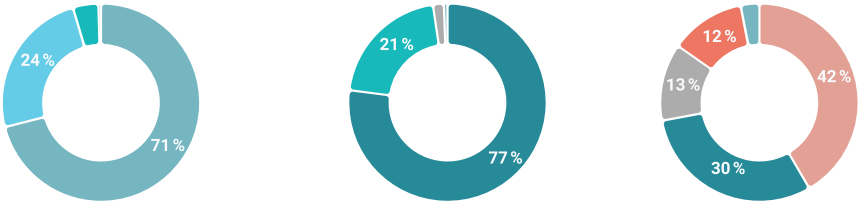


Рис. 5.8 Кольцевые диаграммы с процентными метками

В качестве последнего шага добавим в центр кольцевых диаграмм представляемый ими год. Для этого в каждый контейнер с кольцевой диаграммой добавим текстовый элемент. Поскольку в этот момент мы выполняем итерации по годам, мы можем сразу отобразить текущий год в тексте метки. Кроме того, так как контейнеры кольцевых диаграмм расположены в центре, текстовый элемент автоматически позиционируется правильно. Нам нужно лишь установить его свойства `text-anchor` и `dominant-baseline`, чтобы отцентрировать текст по горизонтали и вертикали. Полученный результат показан на рис. 5.9:

```
donutContainer
  .append("text")
    .text(year)
    .attr("text-anchor", "middle")
    .attr("dominant-baseline", "middle")
    .style("font-size", "24px")
    .style("font-weight", 500);
```

Вот и все! Наши кольцевые диаграммы готовы!

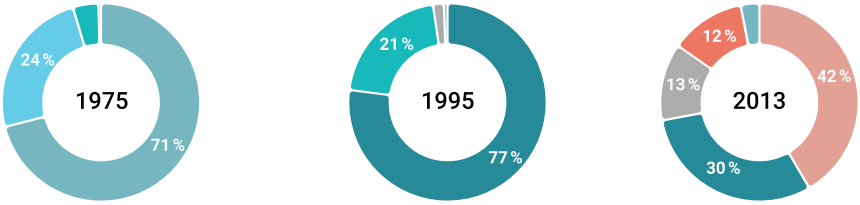


Рис. 5.9 Кольцевые диаграммы с метками годов

Шаги по созданию круговой или кольцевой диаграммы перечислены на рис. 5.10. На первом этапе мы выполняем предварительную обработку данных с помощью функции макета `d3.pie()` и получаем аннотированный набор данных с углами для каждого сегмента. Затем рисуем дуги с помощью функции генератора дуг, которая извлекает углы из аннотированного набора данных и возвращает атрибут `d` для элемента `path`. Наконец, добавляем метки для улучшения читаемости диаграммы, используя текстовые элементы SVG.

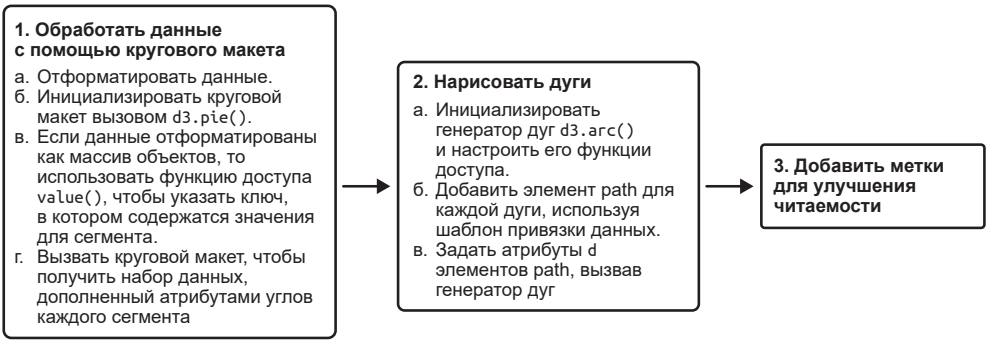


Рис. 5.10 Основные этапы создания круговой или кольцевой диаграммы

5.2 Создание стековых диаграмм

До сих пор мы исследовали довольно простые примеры визуализации информации, которые легко создать в любом программном обеспечении электронных таблиц. Но вы взялись за эту работу не для того, чтобы делать диаграммы в Excel. Мы полагаем, что вы горите желанием поразить свою аудиторию красивыми визуализациями, выиграть награды за эстетику и вызвать глубокий эмоциональный отклик.

Потоковый график – это уже высший пилотаж в визуализации информации, он представляет вариацию и изменение. Его создание может показаться сложным, пока вы не соберете части графика воедино. В конечном счете потоковый график – это разновидность известной составной столбчатой диаграммы. Слои накладываются друг на друга и регулируют площадь элементов выше и ниже, основываясь на пространстве, занимаемом компонентами ближе к центру. График выглядит органичным, потому что имитирует способ роста многих организмов в природе и, по-видимому, подразумевает виды свойств, которые управляют ростом и распадом организмов. Его интерпретацией мы займемся позже, а пока выясним, как его построить.

Мы видели потоковый график в первом разделе этой книги, потому что на самом деле он не такой уж и экзотический. Потоковый график – это стековая диаграмма, т. е. в своей основе он похож на стековые столбчатые диаграммы, как показано на рис. 5.11. Потоковые графики также похожи на область размаха позади линейной диаграммы, которую мы создали в главе 4, за исключением того, что в потоковом графике эти области накладываются друг на друга. В этом разделе мы воспользуемся генераторами стека и областей размаха и с их помощью создадим стековую столбчатую диаграмму, за которой следует потоковый график.

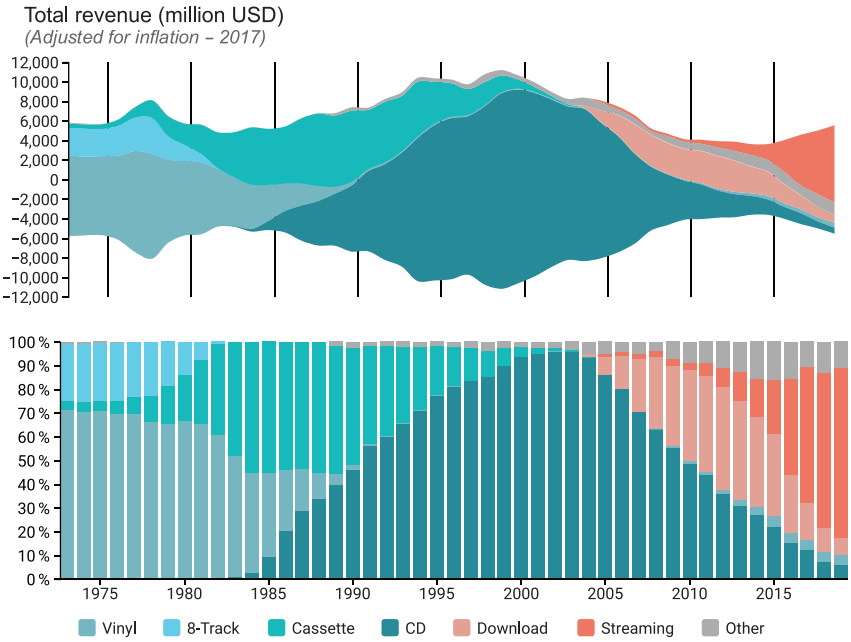


Рис. 5.11 Потоковые диаграммы похожи на стековые столбчатые диаграммы. И те, и другие создаются с помощью генератора стековых макетов

В D3 шаги создания стековой столбчатой диаграммы или потокового графика аналогичны, как показано на рис. 5.12. Сначала мы инициализируем генератор стекового макета и задаем его параметры. Затем передаем в генератор исходный набор данных и получаем новый аннотированный набор данных, определяющий нижнюю и верхнюю границы каждой точки данных. Для создания потокового графика также нужно инициализировать генератор областей размаха, похожий на генераторы линий и кривых, обсуждавшиеся в предыдущей главе. Наконец, мы привязываем аннотированный набор данных к формам SVG, образующим диаграмму, то есть к прямоугольникам для стековой столбчатой диаграммы или элементам `path` потокового графика. Для рисования потокового графика вызывается генератор областей, вычисляющий атрибут `d` элементов `path`. Все эти шаги мы подробно рассмотрим в следующих подразделах.

5.2.1 Генератор стекового макета

Генератор стекового макета – это функция D3, которая принимает набор данных с несколькими категориями. В этом примере мы используем набор данных, содержащий объемы продаж музыки на разных носителях за каждый год с 1973 по 2019 год. Каждый тип носителей будет представлен как серия на стековой диаграмме.

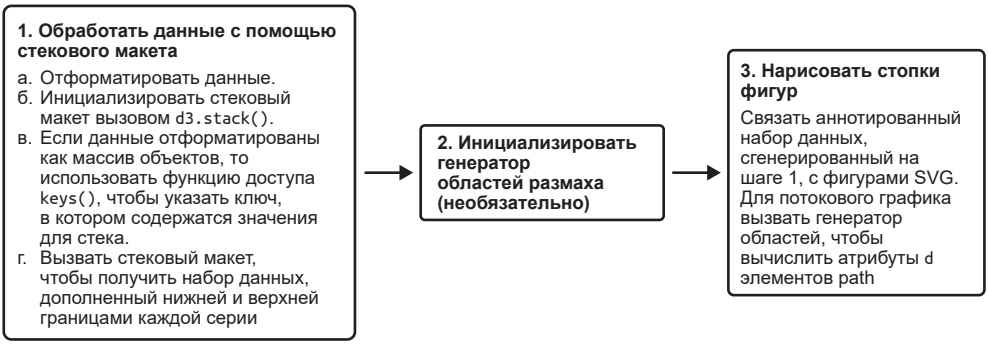


Рис. 5.12 Шаги создания составной столбчатой диаграммы с помощью D3

Как и генератор кругового макета, обсуждавшийся выше, функция стекового макета возвращает новый, аннотированный набор данных, содержащий координаты различных серий, «сложенных» одна над другой. Этот генератор является частью модуля d3-shape.

Давайте включим в работу стековый макет и начнем править функцию `drawStackedBars()`, находящуюся в файле `stacked-bars.js`. Обратите внимание, что эта функция уже содержит код, добавляющий контейнер SVG в `div` с идентификатором `"bars"`, а также контейнер группы для внутренней диаграммы. Это все та же стратегия, которую мы использовали в главе 4, следуя соглашению о границах и отступах.

В следующем фрагменте мы объявляем генератор стекового макета вызовом метода `d3.stack()` и сохраняем его в константе с именем `stackGenerator`. Затем сообщаем генератору, какие ключи из набора данных содержат значения для стековой диаграммы (которые станут серией). Для этого вызывается функция доступа `keys()`, которой передается массив идентификаторов категорий – в данном случае идентификатор каждого носителя музыки. Этот массив создается отображением идентификаторов из константы `formatsInfo`. Мы также могли бы использовать ключ `data.columns`, прикрепленный к набору данных, и отфильтровать годы, как сделали это в разделе 5.1.2.

Наконец, мы вызываем генератор стека и передаем ему исходные данные для получения аннотированного набора, который сохраняем в константе с именем `annotatedData`:

```
const stackGenerator = d3.stack()
  .keys(formatsInfo.map(f => f.id));

const annotatedData = stackGenerator(data);
```

Инициализировать генератор стекового макета

Сообщить функции макета, какие ключи в наборе данных следует использовать для создания серии

Вызвать генератор макета и передать ему данные, чтобы получить аннотированный набор. Сохранить полученный набор в константе

Если вывести аннотированный набор данных в консоль, то можно увидеть, что он имеет вид многомерного массива с вложенными подмассивами. Первый из них – массив для каждой серии, как показано на рис. 5.13, с их идентификаторами, доступными через свойство `key`. Далее следует подмассив серий, содержащий еще один набор массивов, по одному для каждого года в наборе данных. Эти последние массивы включают нижнюю и верхнюю границы категории и исходные данные для соответствующего года. Доступ к нижней и верхней границам осуществляется по индексу `d[0]` и `d[1]` соответственно, где `d` соответствует массиву.

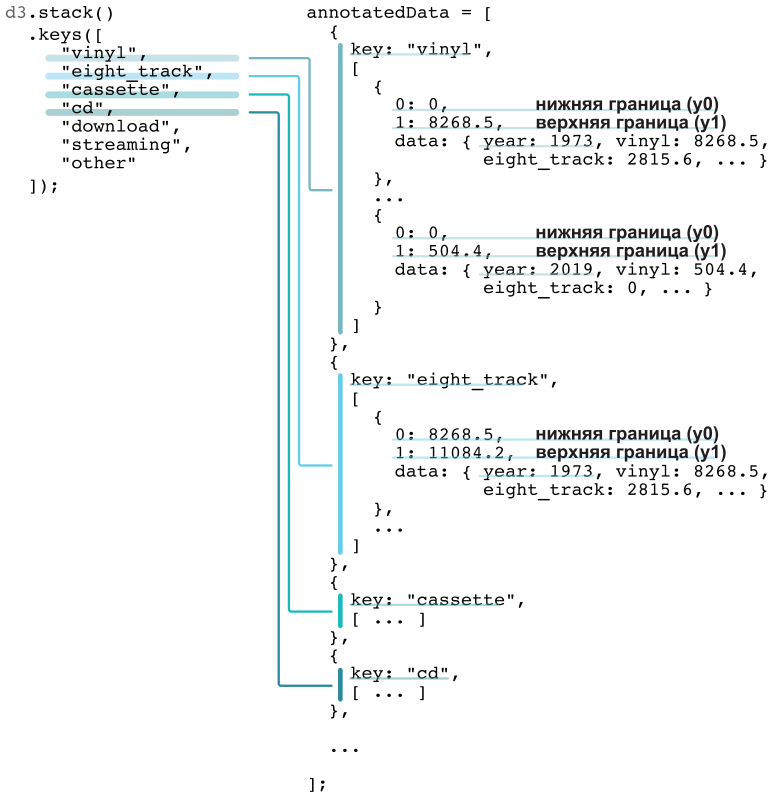


Рис. 5.13 Аннотированный набор данных, возвращаемый генератором стекового макета

Тип носителя `"vinyl"` (виниловые пластинки) – первый ключ, который обрабатывается стековым макетом. Обратите внимание, что его нижняя граница всегда равна 0, а верхняя соответствует продажам носителей этого вида за год. Затем следует тип `"8-track"` (магнитофонная пленка). Его нижняя граница соответствует верхней границе винила, к которой мы добавляем продажи магнитофонных пленок, чтобы получить их верхнюю границу.

Если понятие «стек» вам пока недостаточно ясно, то рис. 5.14 может помочь прояснить ситуацию. Если более подробно рассмотреть данные за 1986 год, то можно увидеть, что музыкальные записи были доступны в продаже в основном на трех носителях: виниловые пластинки с объемом продаж 2,825 млн долл. США, магнитофонные кассеты с объемом продаж 5,830 млн долл. и компакт-диски с объемом продаж 2,170 млн долл. Мы показываем эти точки данных, нарисованные независимо, слева на рис. 5.14.

Используя стековый макет, мы создаем то, что будет называться «столбцами», а не «точками» данных, причем каждый столбец имеет нижнюю и верхнюю границы. Если стек начинается с винила, то нижняя граница равна 0, а верхняя соответствует продажам винила за 1986 год: 2,825 млн долл. Затем мы добавляем сверху продажи кассет, нижняя граница которых соответствует верхней границе винила (2,825 млн), а верхняя является суммой продаж винила и кассет (8,655 млн). Эта верхняя граница становится нижней границей продаж компакт-дисков, а верхняя соответствует сумме продаж трех типов носителей (10,825 млн). Эти границы доступны в аннотированном наборе данных по индексу (`d[0]` и `d[1]`).

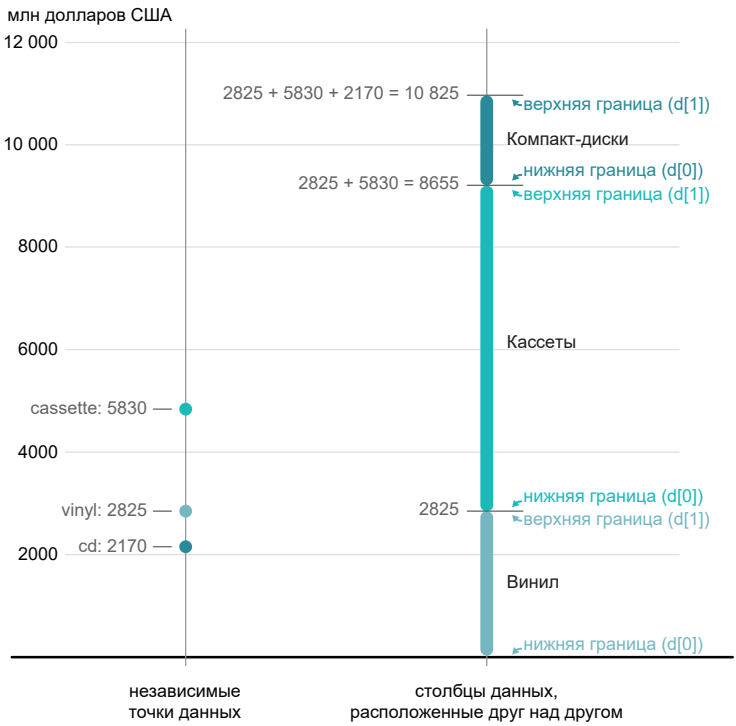


Рис. 5.14 Генератор стекового макета преобразует точки данных в столбцы и возвращает аннотированный набор, содержащий нижнюю и верхнюю границы каждого столбца. Здесь показан пример для 1986 года

5.2.2 Рисование составной столбчатой диаграммы

В этом разделе мы создадим составную столбчатую диаграмму, которую вы видели внизу на рис. 5.11. Составные столбчатые диаграммы похожи на обычные столбчатые диаграммы (гистограммы), которые мы уже создавали в главах 2 и 3, только в данном случае столбцы разделены на несколько категорий, или серий. Составные столбчатые диаграммы и составные визуализации в целом часто используются для отображения развития тенденций с течением времени.

Как и при создании кольцевой диаграммы, мы будем использовать аннотированный набор данных, возвращаемый стековым макетом, чтобы нарисовать столбцы, соответствующие каждой категории. Но сначала определим шкалу для вертикальной оси, с помощью которой будем определять нижнюю и верхнюю границы каждого столбца. Нам нужно, чтобы высота столбцов была прямо пропорциональна продажам, поэтому в листинге 5.6 используется линейная шкала. Поскольку эта шкала должна будет обращаться к аннотированным данным, мы объявим ее внутри функции `drawStackedBars()`.

Предметная область вертикальной шкалы простирается от 0 до максимальной верхней границы, доступной в аннотированных данных. Мы знаем, что это максимальное значение должно находиться внутри последней серии аннотированных данных, которая будет помещена в верхнюю часть диаграммы. Получить доступ к этой серии можно с помощью свойства `length` (`annotatedData[annotatedData.length - 1]`). Затем мы используем метод `d3.max()` для извлечения максимального значения свойства `d[1]`, которое соответствует верхней границе.

Диапазон вертикальной шкалы варьируется от `innerHeight`, нижней части внутренней диаграммы, до 0, верхней части внутренней диаграммы (напомним, что вертикальная ось SVG направлена сверху вниз). Наконец, мы добавляем в цепочку объявления шкалы метод `.nice()`, который гарантирует, что предметная область закончится «красивым» круглым значением, а не фактическим максимальным значением в аннотированном наборе данных.

Листинг 5.6 Объявление вертикальной шкалы (stacked-bars.js)

```
const maxUpperBoundary = d3.max(annotatedData[annotatedData.
➥ length - 1], d => d[1]);

const yScale = d3.scaleLinear()
  .domain([0, maxUpperBoundary])
  .range([innerHeight, 0])
  .nice();
```

Теперь можно добавлять столбцы. Для этого выполним обход аннотированных данных в цикле и добавим серии одну за другой,

как описано в листинге 5.7. Начнем с шаблона привязки данных, который поможет добавить один прямоугольник для каждого года в массив серий (каждому типу носителя соответствует своя серия). Обратите внимание, что к прямоугольникам применяется имя класса, связанное с текущей серией, и затем используется в качестве селектора. Если просто использовать селектор с названием элемента `"rect"`, то в каждой итерации ранее созданные прямоугольники будут удаляться и заменяться новыми.

Далее устанавливаем атрибуты `x` прямоугольников, передавая текущий год в `xScale`, и атрибуты `width`, вызывая свойство `bandwidth` интервальной шкалы. Атрибут `y`, соответствующий вертикальному положению верхнего левого угла прямоугольника, возвращается вертикальной шкалой, объявленной ранее, которой мы передаем верхнюю границу прямоугольника (`d[1]`).

Аналогично высота прямоугольников – это разность между положением их верхней и нижней границ. Здесь есть небольшая заковыка. Поскольку вертикальная ось SVG направлена сверху вниз, `yScale(d[0])` возвращает большее значение, чем `yScale(d[1])`. Нам нужно вычесть последнее из первого, чтобы избежать присваивания отрицательного значения атрибуту `y`, что приведет к ошибке.

Наконец, настраиваем атрибут `fill`, передавая текущий тип носителя в цветовую шкалу, которая доступна в свойстве `key` для каждой серии, как показано ранее на рис. 5.13.

Листинг 5.7 Добавление составных столбцов (stacked-bars.js)

```
annotatedData.forEach(series => {

  innerChart
    .selectAll(`.bar-${series.key}`)
    .data(series)
    .join("rect")
      .attr("class", d => `bar-${series.key}`)

      .attr("x", d => xScale(d.data.year))
      .attr("y", d => yScale(d[1]))
      .attr("width", xScale.bandwidth())
      .attr("height", d => yScale(d[0]) - yScale(d[1]))
      .attr("fill", colorScale(series.key));

});
```

Цикл по сериям в аннотированном наборе данных

Использовать шаблон привязки данных для добавления прямоугольников, соответствующих годам. Для каждой серии создается свое имя класса и используется в качестве селектора, чтобы избежать удаления предыдущих прямоугольников по мере выполнения цикла

Использовать шкалы для размещения прямоугольников и настройки их атрибута fi

Если теперь сохранить проект и открыть его в браузере, то можно заметить, что между столбцами нет горизонтального промежутка. Мы исправим этот недостаток в листинге 5.8, вернувшись к объяв-

лению интервальной шкалы (`xScale`) и добавив вызов функции доступа `paddingInner()` со значением 20 %, как мы делали в главе 3.

Листинг 5.8 Добавление промежутка между столбцами (scales.js)

```
xScale
  .domain(data.map(d => d.year))
  .range([0, innerWidth])
  .paddingInner(0.2);
```

Завершая составную столбчатую диаграмму, добавим оси. В листинге 5.9 мы сначала объявляем нижнюю ось с помощью метода `d3.axisBottom()` и передаем ему шкалу `xScale`.

В цепочку объявления оси мы добавили метод `.tickValues()`, чтобы указать, какие риски и метки должны быть видны на диаграмме. Если этого не сделать, то D3 автоматически добавит риски и метки для каждого года, что из-за стесненности сделает график трудным для чтения. Метод `.tickValues()` принимает массив значений, который мы генерируем с помощью метода `d3.range()` и указываем, что нам нужно каждое целое число с 1975 по 2020 год с шагом 5.

Мы также скрываем риски на концах нижней оси с помощью метода `.tickSizeOuter()`, которому передаем значение `0`. Методы `tickValues()` и `tickSizeOuter()` находятся в модуле d3-axis, а `d3.range()` – в модуле d3-array. Наконец, мы добавляем нижнюю ось с помощью метода `call()` внутри группы, смещенной вниз, и делаем то же самое для левой оси.

Листинг 5.9 Добавление осей (stacked-bars.js)

Объявление генератора нижней оси. С помощью метода .tickValues() настраиваются риски и метки на диаграмме, а с помощью метода .tickSizeOuter(0) скрываются внешние риски

```
const bottomAxis = d3.axisBottom(xScale)
  .tickValues(d3.range(1975, 2020, 5))
  .tickSizeOuter(0);
```

Добавить нижнюю ось в группу SVG и сместить ее в нижнюю часть внутренней диаграммы

```
innerChart
  .append("g")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxis);
```

Объявление и добавление левой оси

```
const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
    .call(leftAxis);
```

Сохранив проект и открыв его в браузере, вы обнаружите, что метки осей слишком маленькие. Кроме того, как упоминалось в главе 4, по умолчанию D3 применяет к группе SVG с элементами осей семей-

ство шрифтов `sans-serif`. Это означает, что метки осей не наследуют семейство шрифтов, заданное в проекте. Из CSS-файла `visualization.css` мы можем определить свойства стиля для меток осей с помощью селектора `.tick text`. Ниже показано, как можно настроить свойства `font-family`, `font-size` и `font-weight`:

```
.tick text {
  font-family: 'Roboto', sans-serif;
  font-size: 14px;
  font-weight: 500;
}
```

Теперь составная столбчатая диаграмма будет выглядеть, как показано на рис. 5.15, но пока она далека от диаграммы, изображенной на рис. 5.2 или в онлайн-версии проекта (http://mng.bz/6nVo). Мы продолжим работу над ней чуть позже.

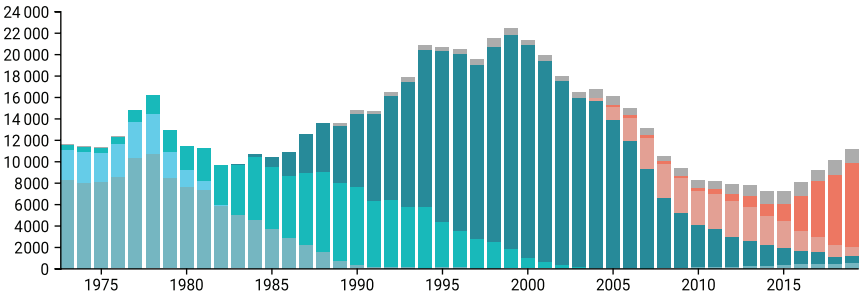


Рис. 5.15 Первая версия составной столбчатой диаграммы

5.2.3 Рисование потокового графика

В предыдущем подразделе мы с помощью функции стекового макета создали аннотированный набор данных, на основе которого нарисовали прямоугольники составной столбчатой диаграммы. Теперь применим похожую стратегию для рисования потокового графика. Потоковые графики выглядят сложнее, чем составные столбчатые диаграммы, но D3 значительно упрощает их создание. Главное отличие в том, что в потоковых графиках аннотированный набор данных используется для создания областей размаха, в то время как в составных столбчатых диаграммах создаются прямоугольники.

В этом подразделе мы будем работать в функции `drawStreamGraph()`, которую вы можете найти в файле `streamgraph.js`. Эта функция уже содержит код, добавляющий контейнер SVG в `div` с идентификатором `"streamgraph"`, а также контейнер группы для внутренней диаграммы. Это все та же стратегия, которую мы использовали в главе 4 вместе с соглашением о границах и отступах.

В листинге 5.10 мы инициализируем генератор стекового макета и вызываем его для получения аннотированных данных. Мы также объявляем линейную шкалу для вычисления местоположения вертикальных границ. Здесь используется тот же код, что и в реализации составной столбчатой диаграммы. Пусть вас пока не беспокоит дублирование кода. Мы решим эту проблему в следующем подразделе.

Листинг 5.10 Объявление генератора стекового макета (streamgraph.js)

```
const stackGenerator = d3.stack()
  .keys(formatsInfo.map(f => f.id));
const annotatedData = stackGenerator(data);
```

Инициализировать и вызвать генератор стекового макета для получения аннотированных данных

```
const maxUpperBoundary = d3.max(annotatedData[annotatedData.
➥ length - 1], d => d[1]);

const yScale = d3.scaleLinear()
  .domain([0, maxUpperBoundary])
  .range([innerHeight, 0])
  .nice();
```

Объявить линейную шкалу для вертикальной оси

Для рисования составных областей размаха нам понадобится функция генератора областей, которая будет отвечать за вычисление значения атрибута `d` элементов `path`, используемых для рисования серий. Как объяснялось в главе 4, генератор областей принимает по крайней мере три функции доступа; в данном случае одна извлекает горизонтальные координаты каждой точки данных, одна – нижние вертикальные границы составных областей и одна – верхние вертикальные границы. Рисунок 5.16 иллюстрирует, как генератор областей применяется к составным областям.

В следующем фрагменте мы инициализируем генератор области `d3.area()`. Сначала используем его функцию доступа `x()` для вычисления горизонтальной координаты каждой точки данных. Поскольку `xScale` – это интервальная шкала, она возвращает координату начала каждого интервала для каждого года, имеющегося в аннотированном наборе данных в объекте `data` каждой точки данных (`d.data.year`). Чтобы выровнять столбцы данных по горизонтали с центрами столбцов в составной столбчатой диаграмме, нужно сместить точки данных вправо на половину ширины столбцов, которую можем вычислить с помощью свойства `bandwidth()` интервальной шкалы.

Затем используем функции доступа `y0()` и `y(1)` для определения вертикальных координат нижней и верхней границ столбцов каждой серии. Эти координаты вычисляются с помощью шкалы `yScale`, объявленной ранее, которой мы передаем значения границ, доступные по их индексам в связанных данных: `d[0]` – для нижней границы и `d[1]` – для верхней.



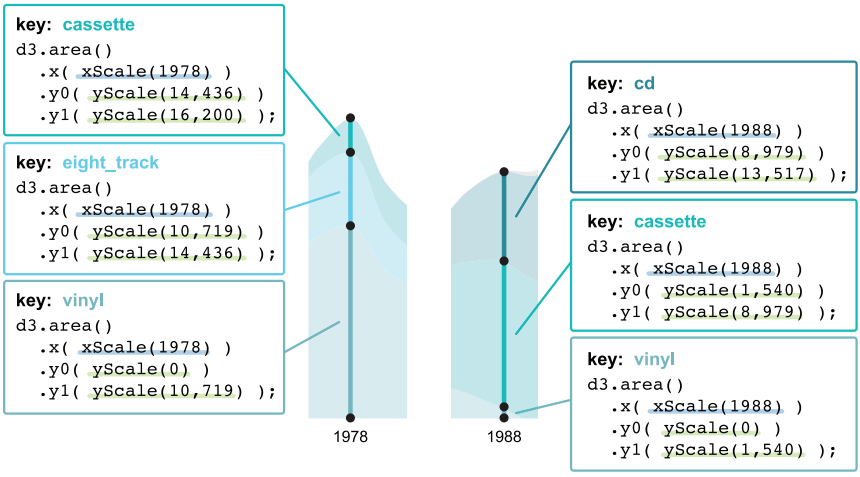


Рис. 5.16 Генератор областей `d3.area()` объединяется с тремя или более функциями доступа. При использовании в сочетании со стековым макетом для создания потокового графа он получает нижнюю и верхнюю границы (`y0` и `y1`) каждой точки данных для вычисления атрибута `d` областей

Наконец, чтобы интерполировать точки данных вдоль каждой границы для получения плавных кривых, а не прямолинейных отрезков, мы используем функцию доступа `curve()`. Здесь мы выбрали функцию интерполяции `d3.curveCatmullRom`:

```
const areaGenerator = d3.area()
  .x(d => xScale(d.data.year) + xScale.bandwidth()/2)
  .y0(d => yScale(d[0]))
  .y1(d => yScale(d[1]))
  .curve(d3.curveCatmullRom);
```

> **ПРИМЕЧАНИЕ** Как подчеркивалось ранее, функции интерполяции влияют на представление данных и должны выбираться с особой тщательностью. Обсуждение и демонстрацию см. в разделе 4.2.2.

Теперь можно начинать рисовать составные области! Сначала используем шаблон привязки данных, чтобы сгенерировать элемент `path` для каждой серии в аннотированном наборе данных. Мы вызываем функцию генератора областей, чтобы получить атрибут `d`

каждого элемента `path`, и вызываем цветовую шкалу для настройки их атрибута `fill`.

Обратите внимание, что элементы `path` добавляются внутри группы SVG, чтобы сохранить разметку организованной и простой для чтения. Позже это также поможет сохранить правильное наложение областей и вертикальной сетки.

```
innerChart
  .append("g")
    .attr("class", "areas-container")
  .selectAll("path")
  .data(annotatedData)
  .join("path")
    .attr("d", areaGenerator)
    .attr("fill", d => colorScale(d.key));
```

Последнее, что мы сделаем в этом разделе, – добавим оси и метки в потоковый график. Начнем с объявления генератора осей `d3.axisLeft()` и передачи ему шкалы `yScale`. Затем добавим элементы осей внутрь группы SVG, используя метод `.call()`:

```
const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
  .call(leftAxis);
```

Превращение рисок на оси в сетку

Мы могли бы опустить ось `x`, учитывая, что потоковый график выровнен по горизонтали с составной столбчатой диаграммой, которая уже имеет ось `x`. Но мы воспользуемся возможностью обсудить, как превратить риски на оси в сетку позади диаграммы.

Прежде всего нужно помнить, что элементы SVG рисуются в порядке их появления в контейнере SVG. Поэтому, чтобы сетка появилась позади потокового графа, ее нужно нарисовать первой. Вот почему следующий фрагмент кода должен находиться *перед* кодом, добавляющим элементы `path` потокового графика.

На данный момент код, генерирующий нижнюю ось, идентичен аналогичному коду составной столбчатой диаграммы, включая использование методов `tickValues()` и `tickSizeOuter()`:

```
const bottomAxis = d3.axisBottom(xScale)
  .tickValues(d3.range(1975, 2020, 5))
  .tickSizeOuter(0);

innerChart
  .append("g")
    .attr("class", "x-axis-streamgraph")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxis);
```

Чтобы преобразовать риски в сетку, нужно всего лишь увеличить их длину с помощью метода `tickSize()`. В данном случае мы задаем длину рисок, равную высоте внутренней диаграммы, умноженной на –1, чтобы они росли вверх:

```
const bottomAxis = d3.axisBottom(xScale)
  .tickValues(d3.range(1975, 2020, 5))
  .tickSizeOuter(0)
  .tickSize(innerHeight * -1);
```

> **ПРИМЕЧАНИЕ** Мы могли бы не перемещать ось вниз и задать длину рисок как положительное значение, чтобы риски росли сверху вниз. Этот же подход можно применить к левой или правой оси, если понадобится нарисовать горизонтальную сетку.

Наконец, мы можем скрыть горизонтальную линию оси и метки годов, задав их непрозрачность равной 0. Для этого используем имя класса, присвоенное контейнеру оси x (`x-axis-streamgraph`), в качестве селектора в CSS-файле `visualization.css`. Как показано в следующем фрагменте, непрозрачность горизонтальной линии, доступ к которой осуществляется через селектор `x-axis-streamgraph path`, управляется свойством `stroke-opacity`, а непрозрачность текстовых меток – свойством `fill-opacity` (селектор `.x-axis-streamgraph text`). Также для управления непрозрачностью в файле `streamgraph.js` можно было бы использовать метод `style()`:

```
.x-axis-streamgraph path {
  stroke-opacity: 0;
}
.x-axis-streamgraph text {
  fill-opacity: 0;
}
```

Сложные текстовые макеты SVG

В качестве последнего штриха добавим метку над левой осью, чтобы показать ее назначение. Как видно на рис. 5.2 выше или в онлайн-версии проекта (http://mng.bz/6nVo), метка потокового графика разбита на две строки: в первой содержится текст «Total revenue (million USD)» (Общий доход (млн долл. США)), а во второй – текст «Adjusted for inflation» (С поправкой на инфляцию).

Мы создадим эту метку с помощью текстового элемента SVG. Имейте в виду, что текст SVG ведет себя не так, как текст HTML. Например, если добавить текст внутрь элемента HTML, то он будет автоматически разбиваться на строки и форматируется в зависимости от доступного пространства по горизонтали. Текст SVG не разбива-

ется и не форматируется, поэтому местоположение каждого текстового элемента необходимо задавать отдельно.

Для управления фрагментами текста внутри SVG можно использовать элемент `tspan`. Разбиение текста на несколько `tspan` позволяет настраивать их стиль и положение по отдельности, используя атрибуты `x`, `y`, `dx` и `dy` – первые два отсчитываются относительно системы координат контейнера SVG, а последние два – относительно предыдущего элемента `text` или `tspan`:

- `x` – горизонтальная координата базовой линии текста относительно системы координат контейнера SVG;
- `y` – вертикальная координата базовой линии текста относительно системы координат контейнера SVG;
- `dx` – смещение базовой линии текста по горизонтали относительно предыдущего текстового элемента;
- `dy` – смещение базовой линии текста по вертикали относительно предыдущего текстового элемента.

Во всех этих определениях важно помнить, что базовая линия контролируется по горизонтали атрибутом `text-anchor`, а по вертикали – атрибутом `dominant-baseline`.

Создать нашу метку можно с помощью трех элементов `tspan`, расположенных внутри текста SVG, как показано на рис. 5.17. Если атрибуту `dominant-baseline` текстового элемента присвоить значение `hanging`, то текст появится непосредственно ниже и правее начала координат контейнера SVG. Используя `dx` и `dy`, можно сместить второй и третий элементы `tspan` в правильные позиции, как показано на рис. 5.17.

```
<text>
  <tspan>Total revenue</tspan>
  <tspan>(million USD)</tspan>
  <tspan>Adjusted for inflation</tspan>
</text>
```

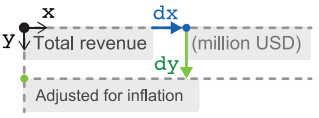


Рис. 5.17 Элементы `tspan` позволяют манипулировать стилем и положением фрагментов текста по отдельности. Атрибуты `dx` и `dy` используются для установки местоположения относительно предыдущего текстового элемента.

Следующий фрагмент кода иллюстрирует реализацию этой стратегии. Сначала мы добавляем текстовый элемент в контейнер SVG и присваиваем его атрибуту `dominant-baseline` значение `hanging`, т. е. базовая линия текста и его дочерних элементов будет располагаться прямо над ними.

Мы сохраняем выбранный текстовый элемент в константу `leftAxisLabel` и повторно используем его для добавления трех элементов `tspan`. В первый элемент `tspan` мы записываем текст `"Total revenue"` (Общий доход), во второй `"(million USD)"` ((млн долл. США)) и в третий `"Adjusted for inflation"` (С поправкой на инфляцию):

```
const leftAxisLabel = svg
  .append("text")
    .attr("dominant-baseline", "hanging");

leftAxisLabel
  .append("tspan")
    .text("Total revenue");

leftAxisLabel
  .append("tspan")
    .text("(million USD)");

leftAxisLabel
  .append("tspan")
    .text("Adjusted for inflation");
```

По умолчанию элементы `tspan` размещаются один за другим на одной горизонтальной линии. Сохраните проект и откройте его в браузере, чтобы убедиться в этом.

Чтобы сместить второй `tspan` немного вправо, можно присвоить его атрибуту `dx` значение `5`. Чтобы сместить третий `tspan` и расположить его ниже первого и второго, можно присвоить атрибуту `y` или `dy` значение `20`. В этом конкретном случае оба атрибута будут иметь одинаковый эффект. Наконец, чтобы выровнять левую сторону третьего `tspan` по левой границе контейнера SVG, лучше всего использовать атрибут `x` и присвоить ему значение `0`:

```
const leftAxisLabel = svg
  .append("text")
    .attr("dominant-baseline", "hanging");

leftAxisLabel
  .append("tspan")
    .text("Total revenue");

leftAxisLabel
  .append("tspan")
    .text("(million USD)")
    .attr("dx", 5);

leftAxisLabel
  .append("tspan")
    .text("Adjusted for inflation")
    .attr("x", 0)
    .attr("dy", 20);
```

Часто элементы `tspan` используются для применения разных стилей к разным частям текста. Например, мы можем уменьшить непрозрачность второго и третьего элементов `tspan`, чтобы придать им серый цвет, и уменьшить размер шрифта третьего `tspan`, поскольку

ку он передает вторичную информацию по сравнению с остальной частью метки:

```
const leftAxisLabel = svg
  .append("text")
    .attr("dominant-baseline", "hanging");

leftAxisLabel
  .append("tspan")
    .text("Total revenue");

leftAxisLabel
  .append("tspan")
    .text("(million USD)")
    .attr("dx", 5)
    .attr("fill-opacity", 0.7);

leftAxisLabel
  .append("tspan")
    .text("Adjusted for inflation")
    .attr("x", 0)
    .attr("dy", 20)
    .attr("fill-opacity", 0.7)
    .style("font-size", "14px");
```

Первая итерация в процессе создания потокового графика завершена, а ее результат показан на рис. 5.18. Когда вертикальная базовая линия такой диаграммы находится на отметке 0, мы часто называем ее составной диаграммой с областями, тогда как потоковые графики, как правило, располагают свои области вокруг базовой линии, пересекающей центр диаграммы. В следующем подразделе мы расскажем, как изменить базовую линию диаграммы. Но прежде посмотрите, насколько похожи на этом этапе составная столбчатая диаграмма и составная диаграмма с областями. Последняя выглядит как сглаженная версия первой.

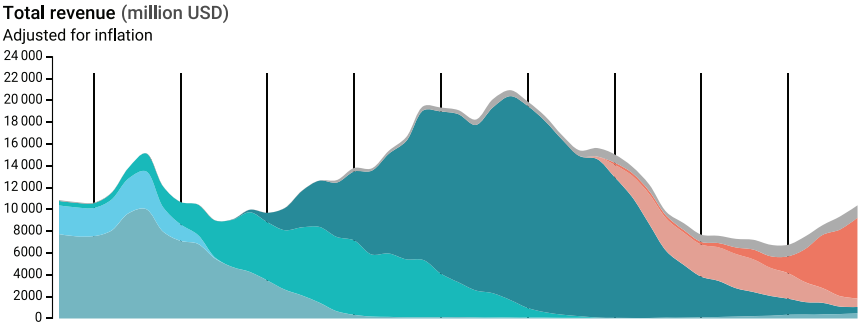


Рис. 5.18 Результат первой итерации в процессе создания потокового графика, который также называется составной диаграммой с областями

5.2.4 Функции доступа order и offset

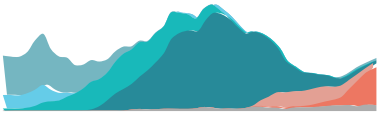
Составная столбчатая диаграмма и составная диаграмма с областями позволяют управлять порядком размещения серий друг над другом и их расположением по вертикали вокруг базовой линии. Такую возможность обеспечивают функции доступа `order()` и `offset()`, которые применяются к генератору стекового макета.

Рассмотрим сначала функцию доступа `order()`, которая управляет порядком расположения фигур по вертикали. В D3 имеется шесть встроенных порядков, которые можно передать функции в виде аргумента, как показано на рис. 5.19.



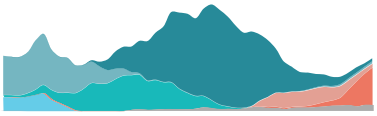
`d3.stackOrderNone`

Серии располагаются снизу вверх в том же порядке, в каком они определены в функции доступа `key()`. Если функция доступа `order()` не используется, то по умолчанию применяется порядок `d3.stackOrderNone`



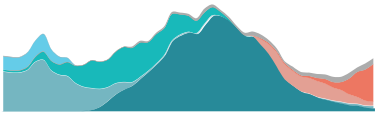
`d3.stackOrderReverse`

Серии располагаются в порядке, обратном указанному в функции доступа `key()`



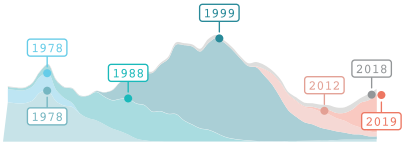
`d3.stackOrderAscending`

Серии располагаются в порядке возрастания общей суммы, начиная снизу с серии с наименьшей общей суммой



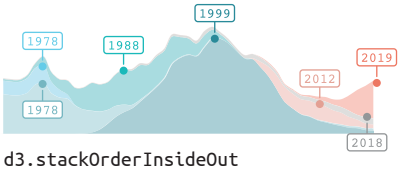
`d3.stackOrderDescending`

Серии располагаются в порядке убывания общей суммы, начиная снизу с серии с наибольшей общей суммой



`d3.stackOrderAppearance`

Отыскивает индексы, в которых каждая серия достигает максимального значения, и располагает серии в порядке возрастания индексов, начиная снизу с серии с самым ранним индексом



`d3.stackOrderInsideOut`

Отыскивает индексы, в которых каждая серия достигает максимального значения. Серия с самым ранним индексом располагается внутри, а серии с более поздними индексами – снаружи. Этот порядок рекомендуется для потоковых графиков вместе с `d3.stackOffsetWiggle`

Рис. 5.19 D3 позволяет управлять порядком размещения фигур друг над другом с помощью функции доступа `order()`. Здесь показаны примеры с областями, но те же принципы применимы и к столбчатым диаграммам

`d3.stackOrderNone` – это порядок по умолчанию; он применяется, если функция доступа `order()` не вызывается. Он располагает фигуры, соответствующие каждой серии, в том же порядке, в каком они

перечислены в массиве ключей, снизу вверх. `d3.stackOrderReverse` меняет этот порядок на обратный, т. е. внизу располагает фигуру, соответствующую последнему ключу, а на самом верху – первому.

`d3.stackOrderAscending` вычисляет общую сумму каждой серии. Серия с наименьшей общей суммой располагается внизу, а остальные – в порядке возрастания снизу вверх. Аналогично `d3.stackOrderDescending` располагает серию с наибольшей общей суммой внизу и остальные серии в порядке убывания снизу вверх.

Последние два порядка отыскивают индексы, в которых каждая серия достигает своего максимального значения. `d3.stackOrderAppearance` располагает серии в том порядке, в каком они достигают своих пиковых значений, что отлично подходит для составных диаграмм с нулевой базовой линией. `d3.stackOrderInsideOut`, напротив, размещает серию с самым ранним пиковым значением в середине диаграммы, а серии с самым поздним пиками – снаружи. Этот порядок отлично подходит для потоковых графиков, где фигуры распределены вокруг центральной базовой линии.

Другая функция доступа стекового макета – функция с именем `offset()` – управляет положением нулевой базовой линии диаграммы и распределением фигур вокруг нее. D3 имеется пять встроенных смещений, как показано на рис. 5.20.

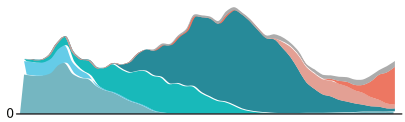
`d3.stackOffsetNone` размещает все фигуры выше нулевой базовой линии. Это смещение по умолчанию. Следующие три смещения распределяют фигуры выше и ниже базовой линии: `d3.stackOffsetDiverging` размещает положительные значения выше базовой линии, а отрицательные – ниже. Это смещение лучше всего подходит для составных столбчатых диаграмм. `d3.stackOffsetSilhouette` смещает базовую линию к центру диаграммы. `d3.stackOffsetWiggle` действует аналогично, но оптимизирует положение базовой линии, чтобы минимизировать покачивание или попеременное движение серий вверх и вниз. Эти три смещения требуют адаптации предметной области вертикальной шкалы для размещения положения базовой линии.

Наконец, `d3.stackOffsetExpand` нормализует значения данных, помещая их в диапазон от 0 до 1 так, чтобы сумма по каждому индексу составляла 100 %. При нормализации значений предметная область вертикальной шкалы также оказывается в диапазоне от 0 до 1.

Для достижения желаемых результатов при создании стековых макетов мы обычно комбинируем порядок и смещение. В целом нет строгих правил, которые указывали бы, когда следует использовать тот или иной порядок или смещение, целью должно быть улучшение читаемости и/или концентрация внимания читателя на аспектах, которые мы пытаемся подчеркнуть.

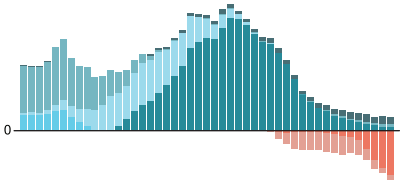
В проекте этой главы мы будем использовать функции доступа `order()` и `offset()` для преобразования составной диаграммы с об-

ластями в потоковый график с центральной базовой линией и в составную столбчатую диаграмму для представления относительных значений (от 0 % до 100 %).



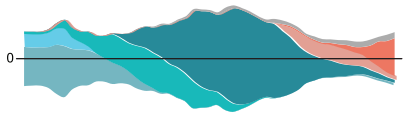
`d3.stackOffsetNone`

Устанавливает нулевую базовую линию и располагает фигуры над ней. Если функция доступа `offset()` не вызывается, то по умолчанию используется `d3.stackOffsetNone`



`d3.stackOffsetDiverging`

Перемещает положительные значения выше базовой линии, а отрицательные значения – ниже. Это смещение лучше всего использовать в составных столбчатых диаграммах



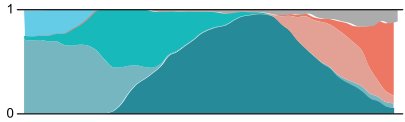
`d3.stackOffsetSilhouette`

Смещает базовую линию в центр диаграммы, распределяя фигуры выше и ниже



`d3.stackOffsetWiggle`

Смещает базовую линию, чтобы минимизировать колебания в каждой серии. Это смещение рекомендуется использовать в паре с `d3.stackOrderInsideOut` при создании потоковых диаграмм



`d3.stackOffsetExpand`

Устанавливает нулевую базовую линию и нормализует значения между 0 и 1

Рис. 5.20 D3 позволяет управлять расположением фигур относительно базовой линии с помощью функции доступа `offset()`. Здесь показаны примеры с составными областями и столбцами

Прежде чем начать, отметим, что функции доступа `order()` и `offset()` могут существенно изменить аннотированный набор данных, возвращаемый генератором макета. Например, превратив составную диаграмму с областями в потоковый график, значение объема продаж будет варьироваться не от 0 до 24 000, а от –12 000 до 12 000. Аналогично если использовать `d3.stackOffsetExpand` для нормализации объемов продаж, отображаемых составной диаграммой, аннотированные данные будут находиться в диапазоне от 0 до 1. Эти различия в значениях необходимо учитывать при выборе предметной области вертикальной шкалы.

Простой способ учесть вариацию предметной области, вызванную применением разных функций доступа `offset()`, – гарантировать вычисление минимального и максимального значений в аннотированном наборе данных и соответственно устанавливать предметную область.

Код в листинге 5.11 начинается с объявления двух пустых массивов, в одном из которых будут храниться минимальные значения серий, а в другом – максимальные. Затем выполняется обход аннотированного набора данных, с помощью `d3.min()` и `d3.max()` определяются минимальное и максимальное значения для каждой серии и помещаются в соответствующие массивы. Наконец, из массивов извлекаются минимальное и максимальное значения и используются для настройки предметной области.

Эту стратегию можно применять как к потоковым графикам, так и к составным столбчатым диаграммам. При создании составной столбчатой диаграммы может потребоваться удалить метод `nice()` из объявления шкалы, чтобы отображать только значения от 0 до 1.

Листинг 5.11 Объявление yScale (stacked-bar.js + streamgraph.js)

```
const minLowerBoundaries = [];
const maxUpperBoundaries = [];

annotatedData.forEach(series => {
  minLowerBoundaries.push(d3.min(series, d => d[0]));
  maxUpperBoundaries.push(d3.max(series, d => d[1]));
});

const minDomain = d3.min(minLowerBoundaries);
const maxDomain = d3.max(maxUpperBoundaries);

const yScale = d3.scaleLinear()
  .domain([minDomain, maxDomain])
  .range([innerHeight, 0])
  .nice();
```

Объявить два пустых массива: один для хранения минимального значения каждой серии, а другой – максимального

Цикл по аннотированному набору данных и поиск минимального и максимального значений в каждой серии. Поместить найденные значения в соответствующие массивы

Извлечь минимальные и максимальные значения из массивов

Использовать минимальное и максимальное значения для установки границ предметной области

Теперь вы сможете свободно применять любой порядок и смещение, и предметная область в `yScale` будет корректироваться автоматически.

Чтобы превратить составную диаграмму с областями в потоковый график, нужно всего лишь связать функции доступа `order()` и `offset()` с генератором стекового макета, объявленным ранее. В этом примере мы используем порядок `d3.stackOrderInsideOut` в сочетании со смещением `d3.stackOffsetSilhouette`. Мы рекомендуем

вам попробовать также другие комбинации, чтобы увидеть, как они влияют на представление данных:

```
const stackGenerator = d3.stack()
  .keys(formatsInfo.map(f => f.id))
  .order(d3.stackOrderInsideOut)
  .offset(d3.stackOffsetSilhouette);
```

> **СОВЕТ** Потоковые диаграммы эстетически привлекательны и заставляют задержать на них внимание. Но их также довольно трудно читать. Они отлично подходят для ситуаций, когда требуется дать обзор эволюции явления с течением времени. Но если вы хотите, чтобы читатель мог читать и сравнивать значения, то лучше выбрать составные или парные столбчатые диаграммы. Подсказки тоже могут помочь улучшить читаемость потоковых диаграмм. Мы создадим одну такую диаграмму в главе 7.

Аналогично изменим составную столбчатую диаграмму, установив для нее смещение `d3.stackOffsetExpand`, чтобы нормализовать объемы продаж в диапазоне от 0 до 1. Также установим порядок `d3.stackOrderDescending`, чтобы показать, насколько мощно доминировали компакт-диски около 2000 года. И снова попробуйте несколько комбинаций и посмотрите, как они меняют фокус истории, передаваемой диаграммой:

```
const stackGenerator = d3.stack()
  .keys(formatsInfo.map(f => f.id))
  .order(d3.stackOrderDescending)
  .offset(d3.stackOffsetExpand);
```

5.3 Добавление легенды в проект

В этом последнем разделе мы посмотрим прием создания легенд с помощью традиционных элементов HTML и опробуем его на практике, добавив цветовую легенду под столбчатой диаграммой. Легенды являются неотъемлемой частью визуализаций данных и помогают читателям интерпретировать то, что они видят. Обычно легенды содержат текст, и мы знаем, что текст SVG не всегда удобен для манипуляций. Взглянув на цветовую легенду, которую мы собираемся создать (рис. 5.21), вы увидите, что она состоит из ряда цветных квадратов и меток. Чтобы создать эту легенду с помощью элементов SVG, потребуется вычислить точное местоположение каждого прямоугольника и текстового элемента. Это вполне возможно, но есть более простой способ.

D3 используется не только для управления элементами SVG; с ее помощью можно создавать любые элементы DOM и манипулировать ими. Это означает, что легенду можно сконструировать из традиционных элементов HTML и использовать CSS для их позиционирования. Обычно такие легенды требуют организации, и для их создания используется неупорядоченный список HTML (`<ul></ul>`). Каждую комбинацию цвета с меткой можно сохранить в элементе `<li></li>`, при этом цвет может содержать один элемент `<span></span>`, а метку – другой, как показано в следующем примере:

```
<ul>
  <li>
    <span> цвет 1 </span>
    <span> метка 1 </span>
  </li>
  <li>
    <span> цвет 2 </span>
    <span> метка 2 </span>
  </li>

  ...

</ul>
```

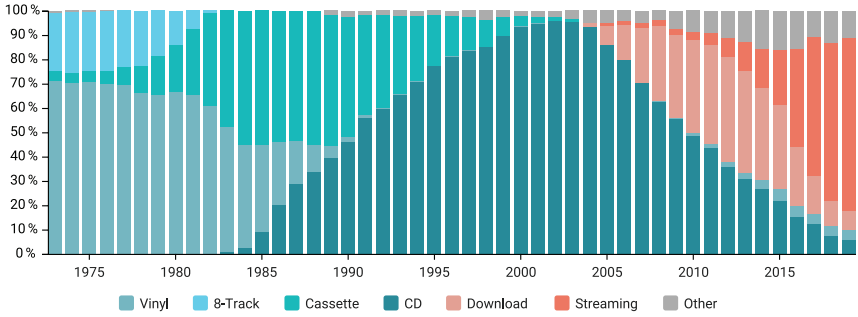


Рис. 5.21 Цветовая легенда расположена под составной столбчатой диаграммой

Чтобы построить эту HTML-структуру с помощью D3, откроем файл `legend.js` и начнем работу в функции `addLegend()`. Код в следующем фрагменте выбирает элемент `div` с классом `legend-container`, который уже существует в `index.html`, добавляет в него элемент `ul` и присваивает ему класс `color-legend`.

Затем используется шаблон привязки данных, чтобы добавить элемент `li` для каждого типа носителя, присутствующего в массиве `formatsInfo` и доступного в `shared-constants.js`. Получившаяся выборка сохраняется в константу с именем `legendItems`.

Далее мы вызываем выборку `legendItems`, добавляем в нее элемент `span` и присваиваем атрибуту `background-color` этого элемента цвет, соответствующий типу носителя. Для этого можно напрямую обратиться к цветовому ключу в `formatsInfo` или вызвать цветовую шкалу. Наконец, мы добавляем еще один элемент `span` и записываем в него ключ метки текущего типа носителя:

```
const legendItems = d3.select(".legend-container")
  .append("ul")
    .attr("class", "color-legend")
  .selectAll(".color-legend-item")
  .data(formatsInfo)
  .join("li")
    .attr("class", "color-legend-item");

legendItems
  .append("span")
    .attr("class", "color-legend-item-color")
    .style("background-color", d => d.color);

legendItems
  .append("span")
    .attr("class", "color-legend-item-label")
    .text(d => d.label);
```

**Создать неупорядоченный список HTML (
) и использовать шаблон привязки данных, чтобы добавить элемент списка (
) для каждого типа носителя**

Добавить элемент span в каждый элемент списка и определить значение его свойства background-color в соответствии с типом носителя

Добавить еще один элемент span в каждый элемент списка и определить его текст в соответствии с меткой типа носителя

Если вы у себя применили те же имена классов, что и мы, то ваша легенда автоматически приобретет вид, как показано на рис. 5.21, потому что следующие стили уже определены в `base.css`:

```
.color-legend {
  display: flex;
  justify-content: center;
  flex-wrap: wrap;
  margin: 0;
  padding-left: 0;
}
.color-legend-item {
  margin: 5px 12px;
  font-size: 1.4rem;
}
.color-legend span {
  display: inline-block;
}
.color-legend-item-color {
  position: relative;
  top: 2px;
  width: 14px;
  height: 14px;
```

```
  margin-right: 5px;
  border-radius: 3px;
}
```

> **ПРИМЕЧАНИЕ** В определении макета легенды используется CSS-свойство `flexbox` (http://mng.bz/qOMz). Мы не будем тратить время на объяснение этого фрагмента стиля, потому что вы, скорее всего, знакомы с каскадными таблицами стилей CSS, обсуждение которых не является целью книги. Главное, что вы должны запомнить: иногда традиционные элементы HTML открывают самый простой путь вперед, и при этом есть возможность использовать D3 для связывания данных и манипулирования любыми элементами DOM.

Теперь вы знаете, как работать с круговыми и стековыми макетами D3! В главе 7 мы добавим интерактивности в этот проект. Вы можете смело переходить прямо туда, если испытываете непреодолимое желание поскорее узнать, как это сделать.

Интервью с Сариной Чен

Чен – специалист по анализу данных.

Расскажите немного о себе и как вы открыли для себя D3.js.

Я получила степень бакалавра по специальности «промышленная инженерия и операционные исследования», а затем перешла в сферу аналитики. Сначала я работала менеджером проектов в консалтинговой фирме, где руководила командами по созданию отчетов для клиентов. Я хотела получить больше практического опыта, поэтому перешла в аналитический стартап. Я много писала на SQL, создавала дашборды Tableau и постепенно оттачивала навыки владения Python.

Мне также было очень интересно заниматься визуальным дизайном. Всякий раз, когда приходилось делать презентации, я представляла все в виде инфографики, чтобы сделать данные более интересными и простыми для понимания. Мой коллега увидел несколько моих виуализаций и порекомендовал познакомиться с D3 и сайтом Надии Бремер (www.visualcinnamon.com). Я была поражена, и мое творческое «я» вдохновилось попробовать D3.

Как вы изучали D3? С какими трудностями столкнулись и как их преодолели?

Я начала с книги *Interactive Data Visualization for the Web* Скотта Мюррея (Scott Murray). Следуя примерам из книги, я также просматривала открытый исходный код на JavaScript из ранних проектов Надии, чтобы закрепить свои знания.

Начав свой первый проект, я часто обнаруживала, что моя творческая натура всегда хотела откусить больше, чем могли прожевать мои технические навыки. У меня вообще нет никакого опыта работы с JavaScript,

поэтому мне приходилось учиться по ходу дела. Даже без объектов, таких как `Promise`, JavaScript кажется странным (для того, кто больше привык к Python).

Более того, функции самой библиотеки D3 выглядели непонятными. Когда я программировала что-то сложнее столбчатой/линейной диаграммы, то просто терпела неудачу. Ничего не отображалось, и я не могла понять, почему. Консоль JavaScript не помогала в отладке, поэтому я много гуглила. Я копировала примеры других людей слово в слово, надеясь, что это сработает. И чем больше я этим занималась, тем больше я узнавала о D3 и привыкала к ней.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашей текущей работе?

Благодаря обширному опыту я свободно владею множеством инструментов, включая Excel, Tableau и Matplotlib (Python). Все они отлично подходят для быстрой проверки идей и анализа, но им не хватает творческой составляющей, которая в избытке имеется в D3!

В своей работе я использую D3 ближе к последнему этапу – большая часть работы, выполняемая до использования D3, заключается в составлении плана визуализации.

Представленный проект: «Holst: The Planets» (https://planets.ifcolorful.com/)



Визуализация пьесы «Меркурий» из симфонической сюиты «Планеты» Густава Холста (Gustav Holst)

Пожалуйста, расскажите кратко о вашем проекте «Holst: The Planets».

Мой проект – визуализация семи частей симфонии «The Planets» Густава Холста. Каждая часть представляет планету и вызывает определенную эмоцию. Например, Венера – это красота, а Юпитер – веселье.

Я начала этот проект, потому что TwoSet Violin – один из моих любимых YouTube-блогеров – разработал креативный мерч на основе этой симфонии. Имея опыт работы с данными в Spotify, я захотела еще раз попробовать заняться визуализацией. Я хотела, чтобы моя графика рассказывала о музыке, подобно движению дирижера, который смотрит на партитуру оркестра и знает все отдельные партии.

Опишите процесс создания этого проекта.

Мой типичный процесс выглядит так:

Сбор данных: опираясь на прошлый опыт, я решила использовать данные анализа аудио с помощью Spotify. Поэтому я повторно использовала свой код для взаимодействий с Spotify API.

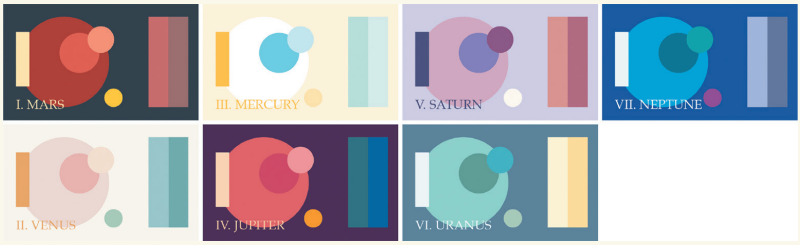
Исследование: для проведения анализа я использовала Pandas и Matplotlib (библиотеки для Python). Я хотела найти интересные измерения (подкрепленные моими музыкальными знаниями) и имела семь разнообразных пьес. Я также выяснила, что не следует делать предположений, основываясь на неполноте или неточности.



Результаты исследований технических особенностей пьес, проведенных Сариной Чен

Выбор сюжета: из всех ракурсов, исследованных на шаге 2, я выбрала основные моменты, которые хотела показать в визуальном ряде. В своей визуализации «The Planets» я хотела описать эмоции и контекст, исходящие от каждой пьесы, и как меняются различные элементы музыки (например, громкость, ритм и т. д.).

Проектирование дизайна: для составления макетов своих диаграмм я использую ручку и бумагу или Illustrator. Мне также было очень интересно использовать Illustrator для выбора семи цветовых палитр, по одной для каждой планеты.



Цветовые палитры, созданные Сариной Чен для каждой планеты

Подготовка данных: узнав, какие данные соответствуют каждому визуальному элементу, я обработала данные в Python и экспортировала их в форматы CSV и JSON для загрузки в D3. Я оставила в коде подробные комментарии, чтобы его можно было легко изменить или вспомнить, что он делает, после долгого перерыва.

Рисование: к этому этапу у меня имеется все необходимое для создания визуального образа и зарезервировано немного времени на небольшие доработки, если встретятся какие-то трудности (например, если пойму, что визуальный образ отлично выглядит на макете, но ужасен на практике).

Расскажите немного о сочетании D3 с Canvas. Какую проблему в этом проекте оно помогло вам решить?

Мне нравится SVG, но этот инструмент не подходит, когда требуется отрисовать слишком много элементов. Поэтому я стараюсь придерживаться проверенного правила: если визуализация содержит более 1000 элементов и вы добавляете движение, то лучше переключиться на Canvas, чтобы не допустить ухудшения производительности. Canvas – это один элемент HTML, и он сможет заставить эту 1000 элементов перемещаться более или менее плавно.

Было несколько проектов, в которых я создавала статическую версию в SVG, а затем, добавляя движение, понимала, что с самого начала следовало использовать Canvas. К счастью, я нашла примеры гибридных подходов, позволяющих выполнять привязку с помощью D3, а рисовать графику с помощью Canvas. Однажды этот метод спас меня от полной переделки кода.

Плавные перемещения, происходящие во время прокрутки, делают этот проект особенно потрясающим! Использовали ли вы еще какую-то библиотеку JavaScript, отличную от D3, для их создания?

Это был мой первый проект с использованием событий прокрутки, поэтому я потратила много времени, исследуя перемещения на основе прокрутки, чтобы выяснить, какую библиотеку применять. Главными моими критериями были простота, удобство использования и релевант-

ность. Чтобы сделать выбор по двум первым критериям, я прочитала несколько статей, написанных опытными специалистами по визуализации данных, которые уже оценили варианты и описали свой личный опыт. Чтобы понять релевантность, я сравнила еженедельные загрузки моих лучших библиотек на NPM (менеджер пакетов JavaScript) и посмотрела, когда был сделан последний коммит на GitHub, дабы узнать, продолжает ли развиваться код.

Я отношусь к библиотеке анимации Scrollama.js как к вишенке на торте, потому что добавляю ее только после реализации всех своих диаграмм в D3. Мои диаграммы D3 и перемещения в них заключены в функции, поэтому библиотеке Scrollama нужно просто вызвать функцию, когда я выполняю прокрутку вверх или вниз.

Что бы вы посоветовали нашим читателям, изучающим D3 и занимающимся созданием своих проектов?

Ставьте цели для каждого личного проекта. Сама я ставила перед собой такие цели, как построение первого сетевого графика, использование метода Scrollytelling (в представленном выше проекте) и создание адаптивной страницы. На творческом фронте у меня еще остаются недостигнутые, например нарисовать нестандартные диаграммы с несколькими кодировками или разработать настраиваемую цветовую палитру.

Итоги

- Макеты D3 – это функции, которые принимают набор входных данных и создают новый аннотированный набор, содержащий атрибуты, необходимые для создания определенной визуализации. Макеты включают этап предварительной обработки, на котором производится форматирование данных и их подготовка к отображению в выбранной форме.
- Круговой макет `d3.pie()` вычисляет начальный и конечный углы каждого сегмента круговой или кольцевой диаграммы:
  - круговой макет предполагает, что входные данные будут иметь вид массива чисел или объектов. Каждый элемент массива соответствует сегменту круговой диаграммы;
  - если данные отформатированы как массив объектов, то мы используем функцию доступа `value()`, чтобы сообщить круговому макету ключ объекта, в котором хранится значение, определяющее размер сегмента;
  - вызывая функцию кругового макета и передавая ей входные данные, мы получаем аннотированный набор данных;
  - аннотированный набор данных содержит начальные и конечные углы каждого сегмента круговой диаграммы;
  - чтобы нарисовать дуги круговой или кольцевой диаграммы, нужно объявить функцию генератора дуг. Она будет исполь-

зовать начальный и конечный углы, содержащиеся в аннотированном наборе данных, для вычисления атрибута `d` SVG-элементов `path`, используемых для рисования дуг.

- Стековый макет `d3.stack()` вычисляет местоположение различных серий при «наложении» их друг на друга:
  - с помощью функции доступа `keys()` мы сообщаем стековому макету, какие ключи из входного набора данных содержат значения для построения диаграммы;
  - вызывая функцию стекового макета, мы передаем ей входные данные и получаем аннотированный набор;
  - аннотированный набор данных содержит значения нижней и верхней границ каждой серии, доступные по индексу (`d[0]` и `d[1]` соответственно). Он также содержит ссылку на входные данные;
  - чтобы нарисовать составную столбчатую диаграмму, мы используем данные, возвращаемые стековым макетом, и добавляем прямоугольники, местоположение которых зависит от нижней и верхней границ каждой серии;
  - чтобы нарисовать потоковый график, мы инициализируем функцию генератора области размаха и используем его функции доступа, дабы указать ему, как получить значения нижней и верхней границ в аннотированном наборе данных. Затем мы используем аннотированный набор данных для добавления SVG-элементов `path` и вычисления их атрибута `d` с помощью генератора области размаха;
  - порядок расположения фигур друг над другом можно контролировать с помощью функции доступа `order()`. D3 предлагает шесть встроенных порядков;
  - размещение фигур вокруг нулевой базовой линии составной диаграммы можно контролировать с помощью функции доступа `offset()`. D3 предлагает пять встроенных смещений;
  - порядки и смещения влияют на предметную область диаграммы, что следует учитывать при настройке масштаба, отвечающего за расчет положения составных фигур.
- Порядковые шкалы в D3 принимают и возвращают дискретные данные. Они отлично подходят для дискретных цветовых шкал, где каждому элементу массива ставится в соответствие определенный цвет.
- Легенды – критически важный элемент визуализаций. Когда легенды содержат множество элементов, то стоит подумать о возможности их создания с помощью традиционных элементов HTML и CSS. Такой подход обычно проще, чем использование фигур SVG и текста.
- Мы можем создавать сложные текстовые макеты SVG, разбивая текст на несколько элементов `tspan`. При позиционировании текста SVG атрибуты `x` и `y` задают положение базовой линии текста относительно начала контейнера SVG, тогда как `dx` и `dy` определяют местоположение относительно предыдущего текстового элемента.

6 Визуализация распределений

В этой главе рассматриваются:

- группировка точек данных в корзины;
- рисование гистограмм;
- рисование гистограмм бок о бок в виде пирамидальных диаграмм для визуального сравнения;
- вычисление квартилей и генерирование диаграмм размаха;
- использование скрипичных диаграмм для сравнения распределений нескольких категорий.

Визуализация распределений – распространенная задача в сфере визуализации данных. Распределения данных используются для оценки частоты появления значений данных в определенном диапазоне или вероятности появления точек данных в диапазоне.

В этой главе мы изучим распределение зарплат специалистов по визуализации данных в Соединенных Штатах. Необходимые нам данные мы возьмем из отчета о состоянии сферы визуализации данных в 2021 году, подготовленного сообществом специалистов по визуализации данных (www.datavisualizationsociety.org). Увидеть, как будет выглядеть проект по окончании, можно на рис. 6.1 и в онлайн-версии, доступной по адресу: http://mng.bz/orvd.

Работу над проектом мы начнем с построения диаграммы, чаще всего используемой для представления распределения данных, –

гистограммы. С ее помощью мы отобразим распределение зарплат 788 респондентов, проживающих в США и занимающихся визуализацией данных. Затем мы сравним зарплаты респондентов, идентифицирующих себя как мужчины и женщины, построив пирамидальную диаграмму и диаграмму размаха. Первая удобна тем, что позволяет сравнить две категории, разместив их бок о бок. Последняя отображает дополнительные сведения, отсутствующие в гистограммах, – квартили и медиану набора данных.

В заключение этой главы мы исследуем распределение доходов людей с различными ролями специализацией, работающих в сфере визуализации данных, таких как аналитики, разработчики и дизайнеры. Мы используем скрипичные диаграммы, показывающие форму распределения для каждой роли, к которым добавим межквартильный размах и среднее значение.

Для построения диаграмм, представленных на рис. 6.1, мы введем понятие корзин. *Корзины* – это группы точек данных, как правило, занимающих диапазоны одинаковой ширины. При создании визуализации распределения с помощью D3 мы предварительно сгруппируем данные в корзины, а затем используем их для построения диаграмм.

Прежде чем начать, перейдите к файлам кода для главы 6. Их можно загрузить из репозитория книги на GitHub, если вы этого еще не сделали (http://mng.bz/Xqjv). В папке `chapter_06` файлы с кодом организованы по разделам. Чтобы начать выполнение упражнений этой главы, откройте в редакторе папку `start` в каталоге `6.1-Binning_data` и запустите локальный веб-сервер. Если вам нужна помощь в настройке локальной среды разработки, то обратитесь к приложению A. Более подробную информацию о структуре папок проекта вы найдете в файле `README`.

> **ВНИМАНИЕ** При работе с примерами кода для этой главы открывайте в редакторе только одну папку, `start` или `end`. Если открыть сразу обе и использовать расширение Live Server для обслуживания проекта, то некоторые пути не будут работать так, как ожидалось.

Папка с данными для проекта содержит CSV-файл, в котором каждая строка соответствует ответу на опрос, проводившийся сообществом специалистов по визуализации данных. Набор данных имеет четыре столбца: уникальный идентификатор (UID), роль респондента (дизайнер, разработчик, журналист и т. д.), пол и диапазон зарплат респондента в долларах США. Участники опроса могли выбрать диапазон шириной 10 тыс. долл. США (10 000–19 999, 20 000–39 999, 40 000–59 999 и т. д.). Респондентов, выбравших вариант «240 000 и более», мы не будем включать в визуализацию.

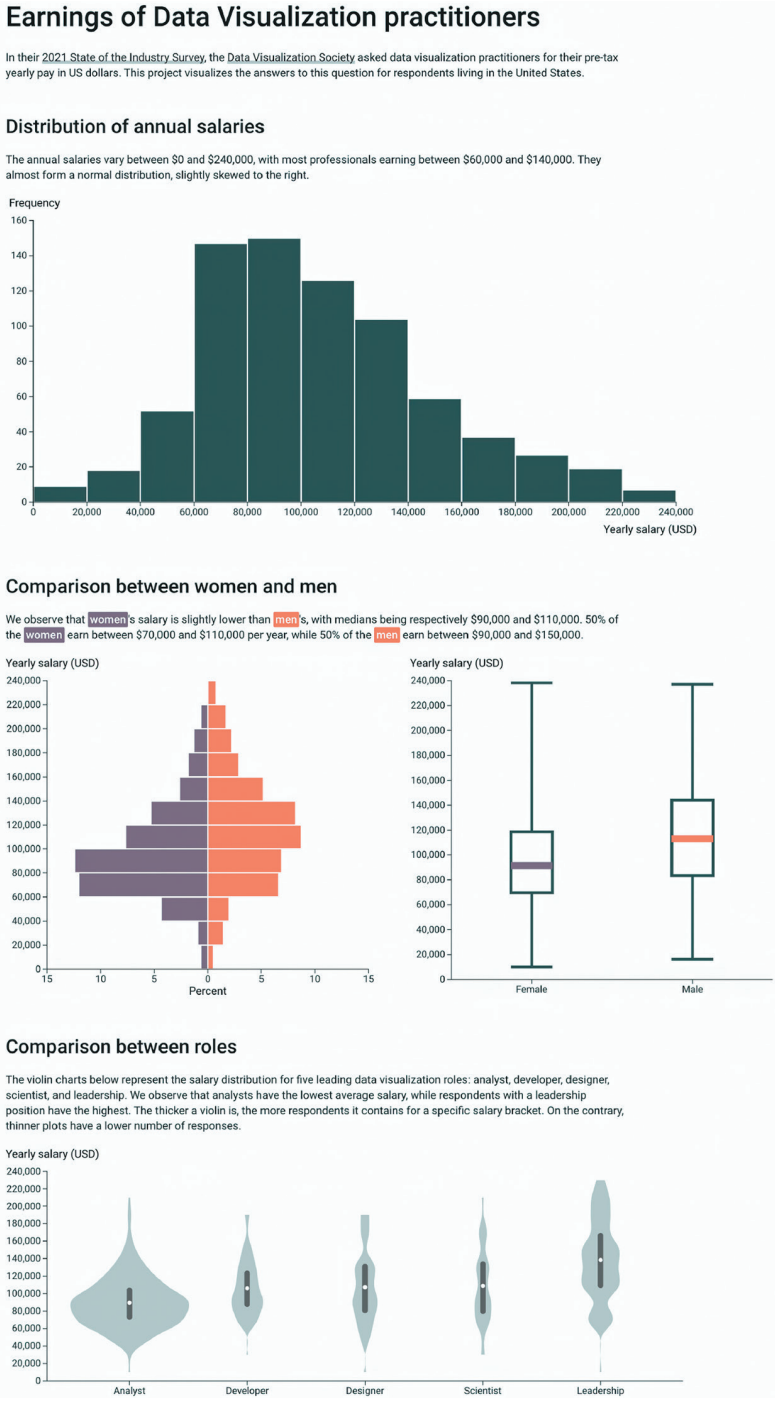


Рис. 6.1 В этой главе мы построим четыре диаграммы для наглядного представления распределения зарплат среди специалистов по визуализации данных в Соединенных Штатах

6.1 Группировка данных

Для визуализации распределения данных обычно необходимо предварительно обработать набор данных, сгруппировав его точки данных в блоки или корзины равной ширины (см. рис. 6.2). В D3 это легко сделать с помощью метода `d3.bin()` из модуля d3-array.

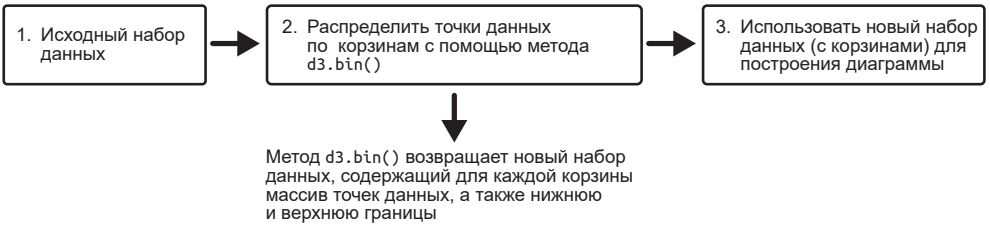


Рис. 6.2 Для визуализации распределения с помощью D3 исходный набор данных передается в функцию `d3.bin()`, которая возвращает новый набор данных, включающий корзины, их границы и соответствующие им точки данных

В качестве примера используем набор данных из проекта для этой главы (`data.csv`). Он содержит информацию о годовой зарплате 788 специалистов по работе с данными, полученную в ходе опроса, в котором респонденту предлагалось выбрать диапазон зарплат. Диапазоны имеют ширину 10 000 или 20 000 долларов США, например 10 000–19 999, 20 000–39 999, 40 000–59 000, 60 000–79 999 и т. д. Каждый диапазон может служить примером использования корзин в визуализации данных. Мы знаем, что фактическая зарплата респондента существует в границах корзины, но не знаем фактического значения.

Проект для этой главы иллюстрирует понятие корзины на примере. Откройте в своем редакторе папку `start`, находящуюся в каталоге `6.1-Binning_data`, и запустите локальный веб-сервер. Открыв файл `loaddata.js`, вы увидите, что набор данных уже загружен в проект с помощью метода `d3.csv()`, обсуждавшегося в главе 3. Соответствующий код показан в листинге 6.1.

Поскольку набор данных с результатами опроса не содержит фактического размера зарплаты каждого респондента, мы вызываем функцию `getRandomSalary()`, чтобы получить случайное целочисленное значение между нижней и верхней границами диапазона зарплат. Мы бы не поступили так в реальном проекте, но здесь такой прием поможет нам проиллюстрировать особенности работы с реалистичными распределениями.

> **ВНИМАНИЕ!** Поскольку для генерации значений зарплат используется функция `Math.random()`, результаты у вас могут отличаться от представленных в этой главе. Но суть разработки таких визуализаций от этого не меняется.

В процессе загрузки данных мы отфильтровываем зарплаты выше 240 000, поскольку мы не знаем верхний предел этого диапазона. После генерирования случайных зарплат и загрузки данных мы вызываем функции `drawHistogram()`, `drawBoxplot()`, `drawPyramid()` и `drawViolinCharts()` – в каждой из них мы построим соответствующие визуализации. Эти функции принимают набор данных и уже объявлены в соответствующих файлах JavaScript.

Листинг 6.1 Загрузка и форматирование набора данных (load-data.js)

Функция getRandomSalary() принимает строку диапазона зарплат, например «$20,000–$39,999», определяет нижнюю и верхнюю границы диапазона и преобразует их в числа, затем возвращает случайное целое число между нижней и верхней границами

```
const getRandomSalary = (salary) => {
  const lowerLimit = +salary.slice(1, salary.indexOf(" -"))
    ➥ .replace(",","");
  const upperLimit = +salary.slice(
    salary.indexOf(" $") + 2)
    ➥.replace(",", "");

  return Math.floor(Math.random() * (upperLimit -
    lowerLimit) + lowerLimit);
}

d3.csv("./data/earnings_per_role.csv", d => {
  if (d.pay_annual_USD !== "$240,000 or more") {
    return {
      role: d.role,
      gender: d.gender,
      salary: getRandomSalary(d.pay_annual_USD)
    };
  }
}).then(data => {

  drawHistogram(data);
  drawBoxplot(data);
  drawPyramid(data);
  drawViolinCharts(data);

});
```

Загрузить набор данных с помощью метода d3.csv()

Отфильтровать диапазоны зарплат «$240,000 or more» с помощью условия if и вернуть специализацию, пол и зарплату респондента. Вызвать функцию getRandomSalary(), чтобы получить случайное значение зарплаты между нижней и верхней границами диапазона

По завершении загрузки данных вызвать функции, конструирующие визуализации, и передать им набор данных. Каждая из этих функций уже объявлена в соответствующем файле JavaScript

Теперь откройте файл `histogram.js`, и начнем работать с функцией `drawHistogram()`. В ней мы уже объявили отступы, ширину и высоту диаграммы. Также, следуя стратегии, описанной в разделе 4.2.1, мы добавили контейнер SVG в DOM и элемент группы SVG, смещенный

в позицию внутренней диаграммы. Эта группа хранится в константе `innerChart`, и позже мы добавим в нее элементы, составляющие гистограмму. То же самое сделано для каждой диаграммы в этой главе.

Далее объявляется генератор корзин вызовом метода `d3.bin()`. Как показано в следующем фрагменте, в цепочке с этим методом можно вызвать несколько функций доступа. Например, поскольку наши данные состоят из массива объектов, мы добавим в цепочку вызов функции `value()`, чтобы сообщить D3 ключ, под которым хранятся значения для группировки. В нашем проекте это ключ `salary`. Наконец, мы генерируем корзины, вызывая генератор и передавая ему набор данных. Обратите внимание, как сильно этот процесс похож на генераторы фигур, обсуждавшиеся в главе 4:

```
const binGenerator = d3.bin()
  .value(d => d.salary);
const bins = binGenerator(data);
```

В последнем фрагменте кода мы сохраняем массив, возвращаемый генератором корзин, в константе с именем `bins`, которую позже мы используем для построения гистограммы. Если вывести содержимое корзин в консоль, то можно увидеть, что они организованы как многомерные массивы (рис. 6.3). Каждый элемент верхнего уровня в массиве корзин содержит массив точек данных, принадлежащих этой конкретной корзине. Свойство `length` корзины сообщает, сколько точек данных она содержит. Нижняя и верхняя границы корзины хранятся в свойствах `x0` и `x1`.



Рис. 6.3 В многомерном массиве, возвращаемом генератором корзин, каждая корзина представлена массивом, содержащим точки данных. На этом рисунке точки данных (объекты JavaScript) для краткости представлены как `{...}`. Нижняя и верхняя границы корзины доступны как свойства `x0` и `x1`

Для иллюстрации идеи корзин мы нарисовали на рис. 6.4 точки данных в соответствующих корзинах. Каждая точка данных представлена кружком, горизонтальное положение которого соответствует величине зарплаты, а ее вертикальное положение выбрано

произвольно, чтобы уменьшить перекрытия между кружками. Как видите, наибольшая плотность точек данных наблюдается между $60K и $140K, а по мере продвижения к краям плотность уменьшается. Это явление также можно будет видеть в визуализациях распределения, которые мы создадим позже. Рисунок 6.4 не является традиционным способом визуализации распределений, но он может помочь вам лучше понять концепцию корзин.

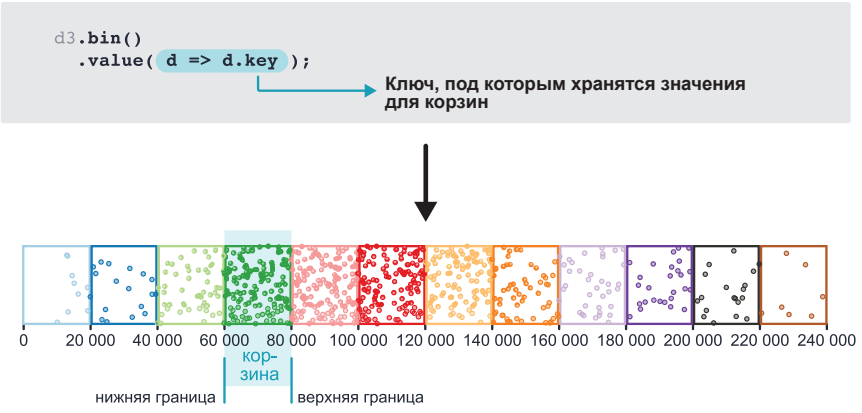


Рис. 6.4 Точки данных с размерами зарплат в соответствующих корзинах. Горизонтальное положение каждого кружка соответствует величине зарплаты, а их вертикальные местоположения выбраны произвольно, чтобы уменьшить перекрытие

6.2 Рисование гистограммы

Гистограммы помогают получить представление о распределении значений в наборе данных, где они сконцентрированы и есть ли заметные разрывы между ними. В этом разделе мы построим гистограмму, отображающую распределение зарплат 755 практикующих специалистов.

Завершенную гистограмму можно увидеть на рис. 6.1 и в онлайн-версии проекта (http://mng.bz/orvd). Обратите внимание, что гистограммы состоят из простых прямоугольников, по одному для каждой группы зарплат. Чтобы построить эту диаграмму с помощью D3, нам нужно лишь добавить элементы прямоугольников в контейнер SVG с шаблоном привязки данных, находящихся в корзинах, полученных в предыдущем разделе. Затем мы должны вычислить местоположение каждого прямоугольника с помощью шкал D3 и установить их высоту, исходя из количества точек данных, которые они представляют, как объясняется на рис. 6.5.

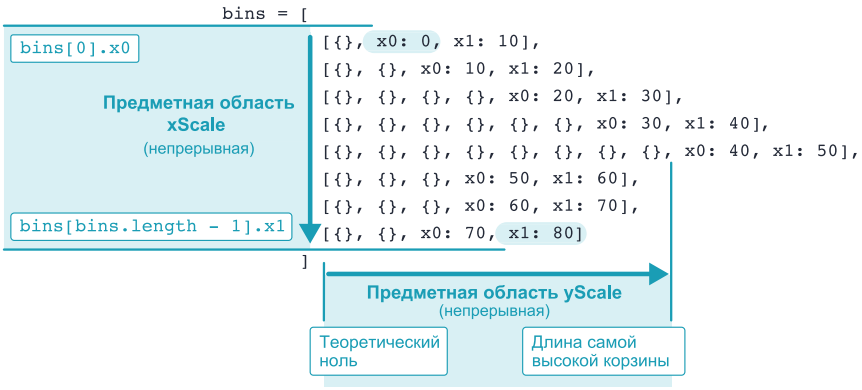


Рис. 6.5 Чтобы сгенерировать гистограмму, мы сначала вызываем метод `d3.bin()`, выполняющий предварительную обработку данных. Он возвращает новый набор, в котором точки данных распределены по корзинам (массивам), и каждая корзина имеет нижнюю и верхнюю границы. Затем мы используем новый набор данных для добавления прямоугольников в контейнер SVG. Длина прямоугольников пропорциональна количеству точек данных, содержащихся в соответствующей корзине, а их местоположение соответствует границам корзин

Давайте начнем с объявления шкал для гистограммы. Как показано на рис. 6.6, нам понадобятся две шкалы: одна для размещения прямоугольников по горизонтали, которую мы назовем `xScale`, и одна для вычисления высоты и размещения прямоугольников по вертикали, которую назовем `yScale`. Поскольку предметные области и диапазоны обеих шкал непрерывны, нам нужно, чтобы диапазоны были линейно пропорциональны предметным областям, поэтому используем линейные шкалы.

Продолжая работать в функции `drawHistogram()` в файле `histogram.js`, объявим горизонтальную и вертикальную шкалы, как подробно описано в листинге 6.2. Предметная область горизонтальной шкалы простирается от минимальной до максимальной зарплат, охватываемых корзинами. Их можно найти, взяв нижнюю границу первой корзины (`bins[0].x0`) и верхнюю границу последней корзины (`bins[bins.length -1].x1`). Диапазон этой шкалы простирается от 0 до `innerWidth`, охватывая всю ширину внутренней диаграммы. Сохраним эту шкалу в константе с именем `xScale`.

С другой стороны, вертикальная шкала отвечает за позиционирование и масштабирование прямоугольников. Ее предметная область простирается от 0 до количества точек данных, содержащихся в самой высокой корзине (`d3.max(bins, d=>d.length)`), а диапазон охватывает значения между `innerHeight` (высотой внутренней диаграммы) и 0. Поскольку в SVG ось `y` направлена сверху вниз, `innerHeight` соответствует нижней границе прямоугольников гистограммы, а 0 – верхней границе диаграммы. Мы сохраняем эту шкалу в константе с именем `yScale` и добавляем в цепочку вызов метода `nice()`, чтобы гарантировать, что ось `y` заканчивается округленным значением.



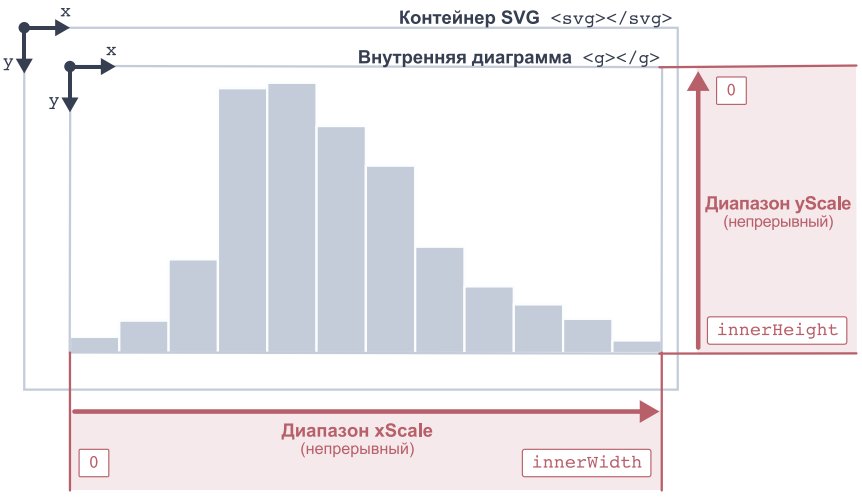


Рис. 6.6 Горизонтальная шкала отвечает за размещение прямоугольников вдоль оси x, а вертикальная шкала располагает прямоугольники вдоль оси y и вычисляет их высоту

Листинг 6.2 Объявление шкал (histogram.js)

Минимальная зарплата на гистограмме соответствует нижнему пределу первой корзины (x0). Максимальная зарплата – верхнему пределу последней корзины (x1)

```
const minSalary = bins[0].x0;
const maxSalary = bins[bins.length - 1].x1;
const xScale = d3.scaleLinear()
  .domain([minSalary, maxSalary])
  .range([0, innerWidth]);
```

Предметная область горизонтальной шкалы простирается от минимальной до максимальной зарплаты, а диапазон – от 0 до innerWidth, что соответствует ширине внутренней диаграммы

Предметная область вертикальной шкалы простирается от 0 до максимального количества точек данных в корзине, а диапазон – от innerHeight до 0; напомним, что ось y направлена сверху вниз

```
const binsMaxLength = d3.max(bins, d => d.length);
const yScale = d3.scaleLinear()
  .domain([0, binsMaxLength])
  .range([innerHeight, 0])
  .nice();
```

Самый высокий прямоугольник соответствует корзине с наибольшим количеством точек данных, которое определяется с помощью d3.max()

Следующий шаг – добавить в диаграмму по одному прямоугольнику для каждой корзины. Для этого к `innerChart` применяется шаблон привязки данных в корзинах, вычисленных в разделе 6.1, после чего D3 предлагается добавить элемент прямоугольника для каждой из них. Как показано на рис. 6.7, атрибуты `x` и `y` прямоугольников соответствуют положению их верхнего левого угла, а атрибуты `width` и `height` определяют их размеры.

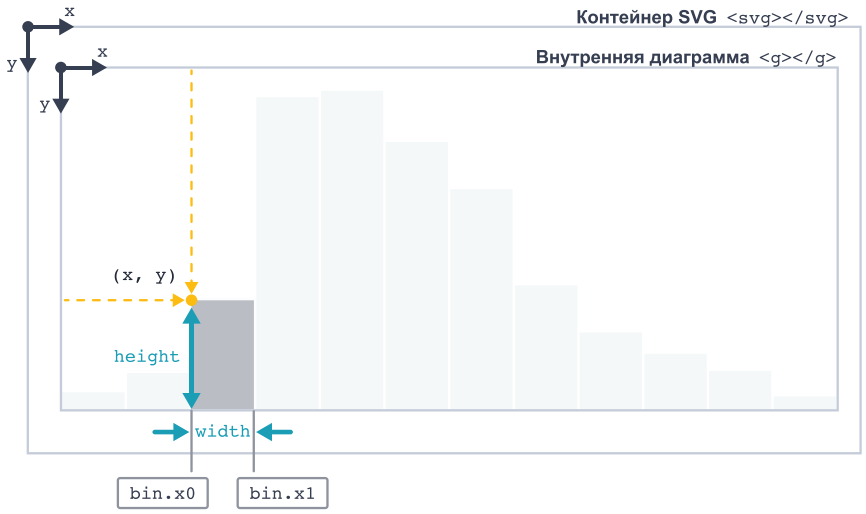


Рис. 6.7 Местоположение прямоугольников гистограммы контролируется их атрибутами `x` и `y`, а размеры – атрибутами `width` и `height`. Высота прямоугольников пропорциональна количеству точек данных, содержащихся в соответствующих им корзинах

В листинге 6.3 мы используем шаблон привязки данных, чтобы добавить по одному прямоугольнику для каждой корзины внутри выборки `innerChart`. Затем настраиваем атрибуты местоположения прямоугольников:

- атрибут `x` каждого прямоугольника соответствует местоположению его нижней границы и вычисляется с помощью `xScale`;
- значение для атрибута `y` определяется передачей свойства `length` корзины в `yScale`;

- ширина прямоугольников определяется как разность между верхней и нижней границами их корзин;
- высота прямоугольников определяется как разность между высотой `innerChart` и местоположением по вертикали их верхнего левого угла.

В завершение атрибуту `fill` прямоугольников присваивается значение переменной `slateGray`, объявленной в файле `shared-constants.js`. Мы также добавляем белую рамку к прямоугольникам, чтобы создать иллюзию пространства между ними.

Листинг 6.3 Добавление прямоугольников (histogram.js)

```
innerChart
  .selectAll("rect")
  .data(bins)
  .join("rect")
    .attr("x", d => xScale(d.x0))
    .attr("y", d => yScale(d.length))
    .attr("width", d => xScale(d.x1) - xScale(d.x0))
    .attr("height", d => innerHeight - yScale(d.length))
    .attr("fill", slateGray)
    .attr("stroke", white)
    .attr("stroke-width", 2);
```

Использовать шаблон привязки данных, чтобы добавить по одному прямоугольнику для каждой корзины

Установить местоположение и размеры прямоугольников

Настроить внешний вид прямоугольников

В качестве последнего шага добавим оси и метки. Начнем с инициализации конструктора нижней горизонтальной оси с помощью метода `d3.axisBottom()` и `xScale`, как подробно описано в листинге 6.4. Затем добавим группу SVG во внутреннюю диаграмму и сместим ее вниз. Вызовом конструктора оси из этой выборки добавим все элементы, составляющие ось внутри группы: отметки, риски и горизонтальную линию. Наконец, добавим в контейнер SVG текстовый элемент с текстом «Yearly salary (USD)» (Годовая зарплата (USD)), который будет служить главной меткой оси и находиться в правом нижнем углу контейнера.

Аналогично добавим левую ось, используя метод `d3.axisLeft()` и `yScale`. После добавления элементов оси добавим метку с текстом «Frequency» (Частота) в верхний левый угол контейнера SVG.

Листинг 6.4 Добавление осей и меток (histogram.js)

```
const bottomAxis = d3.axisBottom(xScale);
innerChart
  .append("g")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxis);
```

Инициализировать конструктор нижней оси, используя xScale

Добавить группу SVG во внутреннюю диаграмму и сместить эту группу вниз. Вызвать конструктор нижней оси, чтобы сгенерировать элементы, составляющие ось

```
svg
  .append("text")
    .text("Yearly salary (USD)")
    .attr("text-anchor", "end")
    .attr("x", width)
    .attr("y", height - 5);

const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
    .call(leftAxis);
svg
  .append("text")
    .text("Frequency")
    .attr("x", 5)
    .attr("y", 20);
```

Добавить текстовый элемент, служащий меткой для нижней оси. Определить его текст и расположить в правом нижнем углу

Инициализировать конструктор левой оси, используя yScale

Добавить группу SVG во внутреннюю диаграмму и вызвать конструктор левой оси, чтобы сгенерировать элементы, составляющие ось

Добавить текстовый элемент, служащий меткой для левой оси. Определить его текст и расположить в левом верхнем углу

Готовая гистограмма показана на рис. 6.8. В следующей главе мы добавим в эту визуализацию фильтрацию по половому признаку.

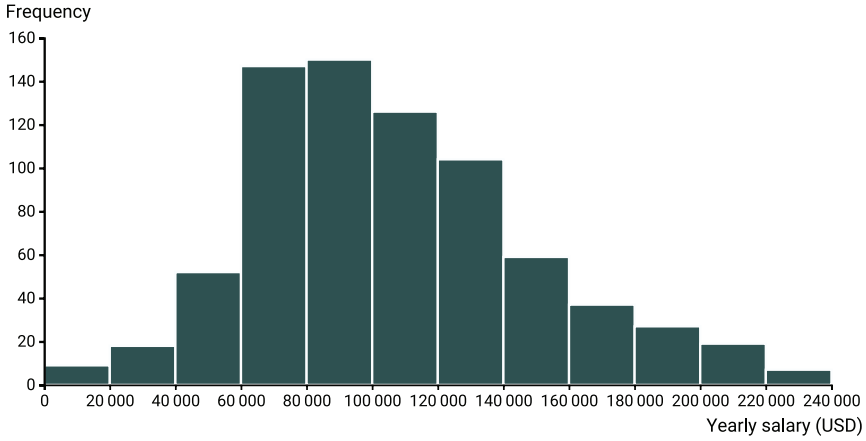


Рис. 6.8 Готовая гистограмма распределения зарплат среди специалистов по визуализации данных в США

6.3 Создание пирамидальной диаграммы

Другой способ визуализации распределений – пирамидальная диаграмма. Она состоит из двух вертикальных гистограмм, расположенных друг напротив друга, и ее легко создать благодаря корзинам D3. Пирамидальные диаграммы используются для сравнения распределения двух категорий.

Часто пирамидальные диаграммы используются для сравнения распределения численности мужчин и женщин по возрастам, как

в примере на рис. 6.9. В таких диаграммах обычно используются прямоугольники, но также можно применять другие формы. Ось x пирамидальных диаграмм часто отражает проценты. Например, на рис. 6.9 показано, что женщины в возрасте от 20 до 24 лет составляют около 3 % от общей численности населения, а мужчины в возрасте от 85 до 89 лет составляют около 1 %.

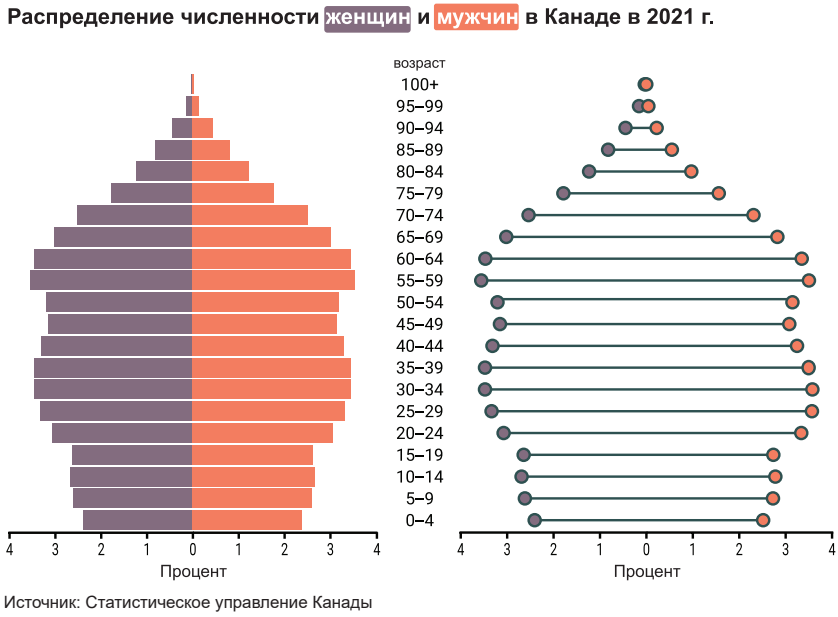


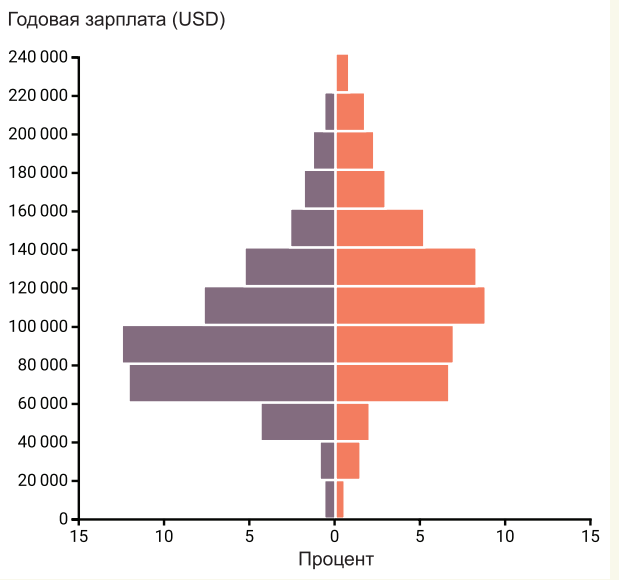
Рис. 6.9 Пирамидальные диаграммы распределения численности мужчин и женщин по возрастам в Канаде в 2021 году. На диаграмме слева для визуализации данных используются прямоугольники, а на диаграмме справа – фигуры в виде отрезков с кружками на концах

Мини-проект: построение пирамидальной диаграммы

Теперь, познакомившись с `d3.bin()` и научившись добавлять прямоугольники в диаграмму, вы сможете самостоятельно построить пирамидальную диаграмму. В этом разделе мы предлагаем вам создать пирамидальную диаграмму, показанную на следующем рисунке, которая отображает распределение размеров заработной платы среди специалистов по визуализации данных в США, идентифицирующих себя как мужчины и женщины.

На этой диаграмме заработки женщин представлены фиолетовыми полосами, а мужчин – оранжевыми. Длина прямоугольников пропорциональна количеству респондентов в группе. Например, мы видим, что 12 % женщин зарабатывают от 60 до 80 тысяч долларов в год, а мужчин

с такой же зарплатой около 6,5 %. Левая половина горизонтальной оси зарезервирована для женщин и простирается от 15 % до 0 %, а правая половина – для мужчин и простирается от 0 % до 15 %. Вертикальная ось представляет зарплаты от 0 до 240 тыс. долларов.



Пирамидальная диаграмма, отражающая распределение размеров заработной платы среди специалистов по визуализации данных в США, идентифицирующих себя как женщин и мужчин

Вот несколько советов, которые могут помочь вам выполнить это упражнение:

- добавляйте свой код в функцию `drawPyramid()` в файле `pyramid.js`. Контейнер SVG и внутренняя диаграмма (группа SVG) уже добавлены внутрь `div#pyramid`;
- сгенерируйте корзины с помощью `d3.bin()`, при этом для мужчин и женщин должны быть сгенерированы отдельные корзины;
- для мужчин и женщин можно объявить разные горизонтальные шкалы. Женская шкала должна охватывать левую половину оси x, а мужская – правую;
- длина столбцов пропорциональна проценту от общего числа респондентов в каждом диапазоне зарплат. Этот процент варьируется приблизительно от 0 % до 15 %;
- каждая сторона пирамидальной диаграммы строится точно так же, как гистограмма, созданная в разделе 6.2. Только имеет другую ориентацию!
- константы со значениями цвета доступны в `shared-constants.js`.

Если у вас возникнут сложности или вы захотите сравнить свое решение с нашим, то вы найдете его в листинге D.2 и в папке `6.3-Pyramid/end` с примерами кода для этой главы. Но мы призываем вас постараться выполнить это упражнение самостоятельно, потому что практика – лучший способ научиться чему-либо!

Обратите внимание, что ваше решение может немного отличаться от нашего, потому что в разработке часто один и тот же результат можно получить разными путями.

6.4 Создание ящичных диаграмм

Ящичные диаграммы (их еще называют коробчатыми диаграммами, или диаграммами размаха) – еще один способ визуализации распределений. Их основная цель – выделить медиану набора данных и проиллюстрировать квартили, делящие точки данных на четыре группы более или менее равного размера. Ящичные диаграммы более компактны, чем гистограммы, но сложнее для понимания читателями, незнакомыми со статистикой.

Как показано на рис. 6.10, ящичная диаграмма состоит из прямоугольника, представляющего 50 % точек данных. Этот прямоугольник охватывает диапазоны от первого квартиля (или 25-го процентиля) до третьего (или 75-го процентиля). Он также пересекается с медианой, то есть порогом, разделяющим нижнюю и верхнюю половины точек данных.

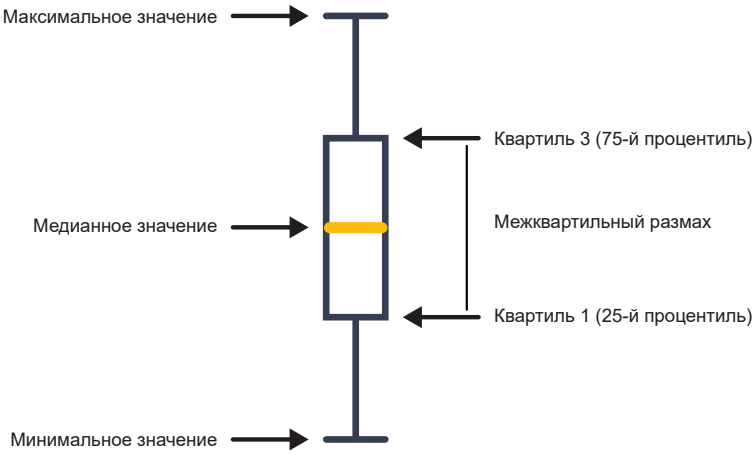


Рис. 6.10 Ящичная диаграмма отображает пять фрагментов информации, закодированных в одной фигуре: минимальное значение в наборе данных, первый квартиль (или 25-й процентиль), медиана (или среднее значение), третий квартиль (или 75-й процентиль) и максимальное значение

Вертикальные линии, идущие от нижней и верхней частей прямоугольника, называют усами. Нижний ус простирается от минимального значения в наборе данных до первого квартиля, а верхний – от третьего квартиля до максимального значения. Выбросы могут изображаться в виде точек за пределами усов.

Создание ящичной диаграммы с помощью D3 начинается с поиска минимальных и максимальных значений в наборе данных, как указано на рис. 6.11. Затем используется квантильная шкала для вычисления квартилей и медианных значений. В заключение в контейнер SVG добавляются прямоугольник и линии и настраиваются их позиционные атрибуты в соответствии с минимальным, максимальным, медианным и квартильными значениями.

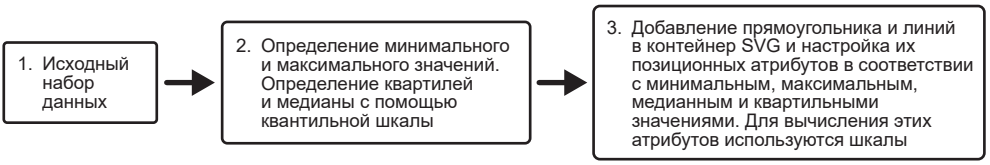


Рис. 6.11 Шаги для создания ящичной диаграммы с помощью D3

6.4.1 Расчет квартилей с помощью квантильной шкалы

Квантили – это точки, разделяющие распределение данных на группы схожих размеров. *Квартили* – это разновидность квантилей, они разделяют данные ровно на четыре группы. Для расчета квантилей и квартилей в D3 используется квантильная шкала (https://d3js.org/d3-scale/quantile).

Вернемся к проекту этой главы и вычислим квартили для ящичных диаграмм, показанных на рис. 6.1 и в онлайн-версии проекта (http://mng.bz/orvd). В этом проекте ящичные диаграммы используются для сравнения зарплат женщин и мужчин, т. е. мы должны визуализировать два набора данных: один с зарплатами женщин и один с зарплатами мужчин. В этом разделе мы будем работать внутри функции `drawBoxplot()`, содержащейся в файле `box-plot.js`. Обратите внимание, что в нее уже включен код, добавляющий контейнер SVG в DOM и группу SVG для внутренней диаграммы, согласно стратегии, представленной в главе 4.

В следующем фрагменте кода мы используем JavaScript-метод `filter()` для выборки зарплат женщин и сохранения их в новом массиве с помощью JavaScript-метода `map()`. Этот массив сохраняется в константе `femaleSalaries`:

```
const femalesSalaries = data.filter(d => d.gender === "Female")
 ➥.map(d => d.salary);
```

Для вычисления квартилей мы используем квантильную шкалу из библиотеки D3, предметная область и диапазон которой являются дискретными. Предметная область – это массив точек данных (в данном случае зарплат женщин или мужчин), а диапазон определяет количество вычисляемых квантилей. Чтобы вычислить квартили, разделив данные на четыре группы, диапазон должен быть массивом из четырех элементов, например `[0, 1, 2, 3]`.

В следующем фрагменте мы объявляем квантильную шкалу с помощью `d3.scaleQuantile()`, передаем массив зарплат женщин в качестве предметной области и массив из четырех значений в качестве диапазона и сохраняем полученную шкалу в константе `femalesQuartilesScale`:

```
const femalesQuartilesScale = d3.scaleQuantile()
  .domain(femalesSalaries)
  .range([0, 1, 2, 3]);
```

Наконец, вычисляем квартили, вызывая функцию доступа `quantiles()` из `femalesQuartilesScale`. Она возвращает пороговые значения квантилей:

```
const femalesQuartiles = femalesQuartilesScale.quantiles();
```

Если сохранить пороговые значения в константе `femalesQuartiles` и вывести их в консоль, то мы получим массив из трех значений:

- первый квартиль, или 25-й процентиль;
- второй квартиль, или 50-й процентиль, также известный как медиана;
- третий квартиль, или 75-й процентиль.

Обратите внимание, что медиану можно также вычислить напрямую с помощью метода `d3.median()` из модуля d3-array. Этот метод принимает массив значений (в данном случае зарплаты):

```
d3.median(femalesSalaries);
```

В листинге 6.5 мы повторяем один и тот же процесс вычислений для зарплат женщин и мужчин, вычисляя минимальные и максимальные значения зарплат с помощью метода `d3.extent()`, который возвращает массив из двух значений: минимума и максимума. Скоро мы используем квартили, минимум и максимум для размещения прямоугольника и линий ящичной диаграммы.

Листинг 6.5 Расчет квартилей и минимальных/максимальных значений (box-plot.js)

Отфильтровать набор данных, сохранив только зарплаты женщин или мужчин, и сохранить оставшиеся значения в новом массиве

```
const femalesSalaries = data.filter(d => d.gender ===
  "Female").map(d => d.salary);
const femalesQuartilesScale = d3.scaleQuantile()
  .domain(femalesSalaries)
  .range([0, 1, 2, 3]);

const femalesQuartiles = femalesQuartilesScale
  .quantiles();
const femalesExtent = d3.extent(femalesSalaries);

const malesSalaries = data.filter(d => d.gender ===
  "Male").map(d => d.salary);
const malesQuartilesScale = d3.scaleQuantile()
  .domain(malesSalaries)
  .range([0, 1, 2, 3]);

const malesQuartiles = malesQuartilesScale.quantiles();
const malesExtent = d3.extent(malesSalaries);
```

Объявить шкалу квантилей для женщин и мужчин. Передать массив зарплат в качестве предметной области и массив из четырех значений в качестве диапазона

Получить массив, содержащий значения квартилей, вызвав функцию quantiles() шкалы квантилей. Найти минимальное и максимальное значения зарплаты с помощью d3.extent()

6.4.2 Позиционирование нескольких ящичных диаграмм на графике

В нашем проекте мы разместим две ящичные диаграммы ящиков на графике, одна будет показывать распределение зарплат женщин, а другая – зарплат мужчин. Как показано на рис. 6.12 и в онлайн-версии проекта (http://mng.bz/orvd), признак пола откладывается на горизонтальной оси, а зарплаты распределяются вдоль вертикальной оси. Это означает, что для создания горизонтальной оси нужна шкала, принимающая дискретные значения предметной области (пол) и возвращающая непрерывный диапазон, т. е. доступное горизонтальное пространство. Для создания вертикальной оси нужна шкала, принимающая непрерывную предметную область (зарплаты) и возвращающая непрерывные значения в пределах вертикальной высоты диаграммы.

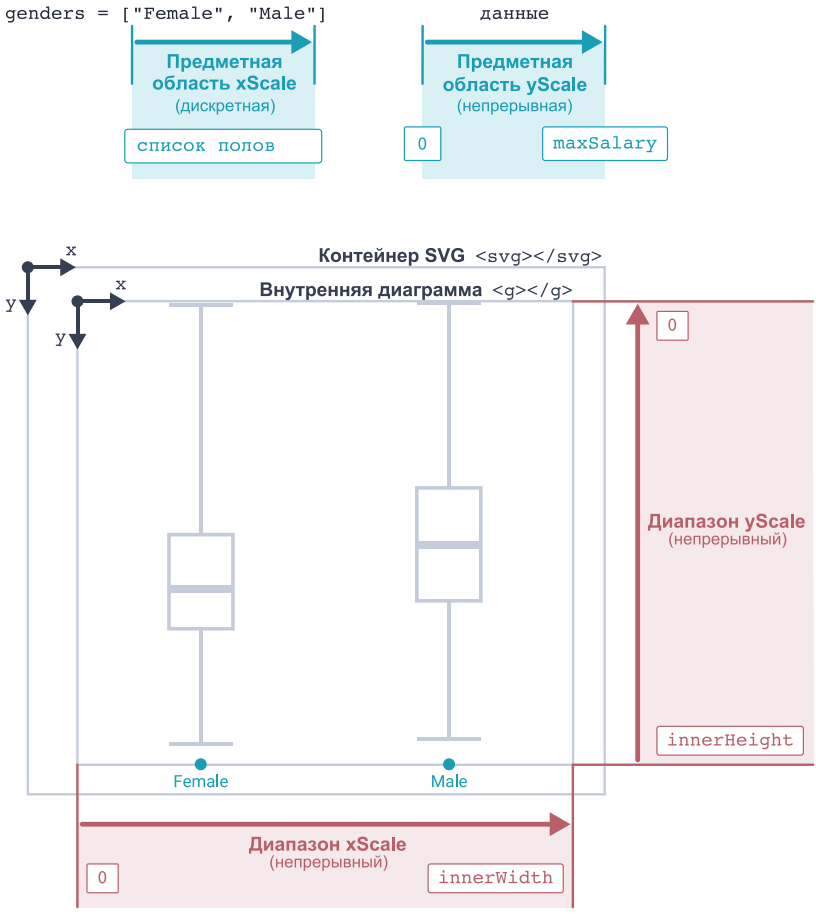


Рис. 6.12 Горизонтальная шкала будет отвечать за размещение ящичных диаграмм для женщин и мужчин вдоль оси x. Вертикальная шкала распределит зарплаты по оси y и поможет вычислить местоположение минимального, максимального, медианного и квартильного значений

6.4.3 Точечная шкала

Для размещения полов по оси x мы используем точечную шкалу `d3.scalePoint()` (https://observablehq.com/@d3/d3-scalepoint). Точечная шкала очень похожа на интервальную шкалу, которую мы использовали в главах 3 и 5, за исключением того, что возвращаемые интервалы равны нулю. Эта шкала используется для распределения дискретных элементов по непрерывному диапазону.

На рис. 6.13 показана точечная шкала, которая в качестве предметной области использует массив букв и доступное горизонтальное пространство в качестве диапазона. Мы связываем шкалу с методом `padding()`, чтобы задать внешний отступ на двух концах. Метод `padding()` принимает коэффициент от 0 до 1. Внешний отступ вы-

числяется как произведение этого коэффициента на размер шагов (`коэффициент отступа * шаг`); шаги представляют промежутки между соседними точками.

```
const pointScale = d3.scalePoint()
  .domain(["A", "B", "C", "D", "E"])   — Список дискретных элементов
  .range([0, innerWidth])              — Начало и конец доступного пространства
  .padding(0.5);                       — Внешний отступ как % от числа шагов
```

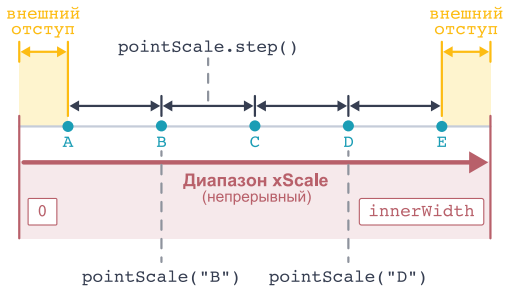


Рис. 6.13 Точечная шкала похожа на интервальную шкалу полосы, в которой интервалы имеют нулевую ширину. Ее предметная область состоит из дискретного списка элементов, распределенных по непрерывному диапазону

В листинге 6.6 перед объявлением шкал мы создаем массив, содержащий две строки, `"Female"` (Женщины) и `"Male"` (Мужчины), и даем ему имя `genders`. Затем объявляем точечную шкалу, которую используем для равномерного распределения полов по оси `x`. В качестве предметной области мы передаем массив `genders`, устанавливаем диапазон от 0 до ширины внутренней диаграммы и задаем отступ, равный 0,5 (50 %) расстояния между позициями двух ящичных диаграмм.

Как и в гистограмме, которую мы построили в разделе 6.2, желательно, чтобы зарплаты были линейно пропорциональны их положению вдоль оси `y`. Чтобы обеспечить это, мы объявляем линейную шкалу. Ее предметная область простирается от 0 до максимальной зарплаты в наборе данных, вычисленной с помощью `d3.max()`, а диапазон – от внутренней высоты диаграммы до 0, поскольку ось `y` направлена сверху вниз. Наконец, мы вызываем метод `nice()` линейной шкалы, чтобы гарантировать, что ось заканчивается круглым значением.

Листинг 6.6 Объявление шкал (box-plot.js)

```
const genders = ["Female", "Male"];
const xScale = d3.scalePoint()
  .domain(genders)
  .range([0, innerWidth])
  .padding(0.5);
```

Объявить точечную шкалу для распределения полов (женщины и мужчины) по оси `x`

```
const maxSalary = d3.max(data, d => d.salary);
const yScale = d3.scaleLinear()
  .domain([0, maxSalary])
  .range([innerHeight, 0])
  .nice();
```

Использовать линейную шкалу для распределения зарплат по оси y. Вызвать ее метод nice(), чтобы гарантировать, что ось заканчивается круглым числом

Затем шкалы используются для рисования осей ящичных диаграмм. В листинге 6.7 мы сначала объявляем конструктор для нижней оси, для чего вызываем метод `d3.axisBottom()` и передаем ему `xScale`. Чтобы скрыть внешние риски на оси, мы добавляем вызов `tickSizeOuter()` (метода из модуля d3-axis, с помощью которого задаются размеры рисок в начале и конце оси), которому передаем значение `0`. Затем добавляем группу SVG во внутреннюю диаграмму, перемещаем ее вниз и вызываем конструктор, чтобы создать элементы оси.

Аналогично объявляется конструктор для левой оси, для чего вызываем метод `d3.axisLeft()` и передаем ему `yScale`. Добавляем группу во внутреннюю диаграмму и вызываем конструктор. Наконец, отображаем метку над осью y, добавив текстовый элемент к контейнеру SVG и присвоив ему текст `"Yearly salaries (USD)"` (Годовые зарплаты (UDS)).

Листинг 6.7 Рисование осей (box-plot.js)

Инициализировать конструктор нижней оси, используя xScale. Установить размеры внешних рисок равными 0

Добавить группу во внутреннюю диаграмму, переместить ее вниз и сгенерировать элементы оси вызовом конструктора

```
const bottomAxis = d3.axisBottom(xScale)
  .tickSizeOuter(0);
innerChart
  .append("g")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxis);

const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
    .call(leftAxis);
svg
  .append("text")
    .text("Yearly salary (USD)")
    .attr("x", 0)
    .attr("y", 20);
```

Инициализировать конструктор левой оси, используя yScale. Добавить группу во внутреннюю диаграмму и сгенерировать элементы оси вызовом конструктора

Добавить метку для оси y с текстом "Yearly salary (USD)"

6.4.4 Рисование ящичной диаграммы

Теперь у нас есть все, что нужно, и мы можем нарисовать наши ящичные диаграммы! Как упоминалось выше, ящичные диаграммы состоят из трех элементов, как показано на рис. 6.14:

- прямоугольника, простирающегося от первого до третьего квартиля;
- линии на уровне медианы, соответствующей второму квартилю;
- усов, или линий, простирающихся от минимального значения до первого квартиля и от третьего квартиля до максимального значения.

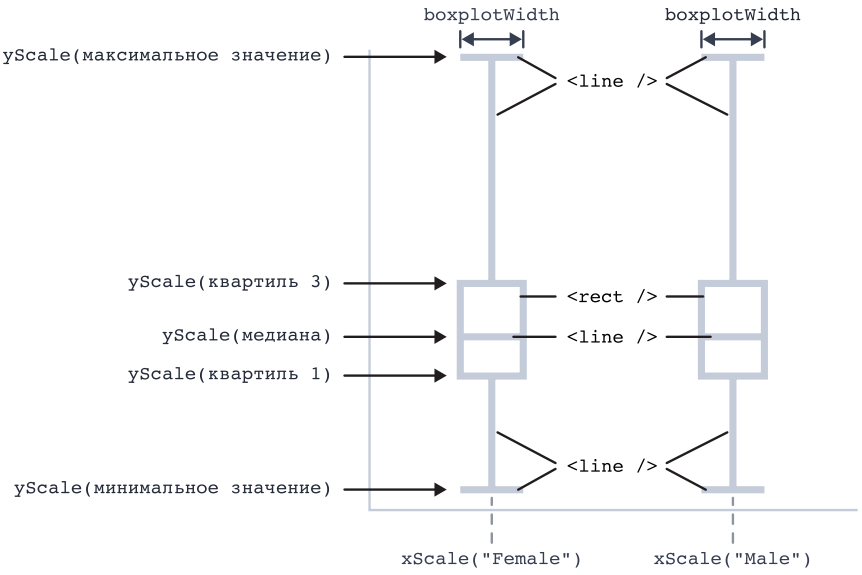


Рис. 6.14 Ящичная диаграмма состоит из прямоугольника, простирающегося от первого до третьего квартиля, линии на уровне медианы и усов, простирающихся от минимального значения до первого квартиля и от третьего квартиля до максимального значения

Рисование ящичных диаграмм мы начинаем с объявления двух констант, `boxplotWidth` и `boxplotStrokeWidth`, в которых хранятся ширина ящичных диаграмм и толщина их рамки. В листинге 6.8 мы даем им значения 60 и 3 пикселя соответственно, но вы можете изменить их по своему усмотрению. Эти значения будут использоваться несколько раз, поэтому хранение их в константах облегчает будущие изменения.

Затем мы выполняем обход элементов массива `genders`, объявленного в листинге 6.8. Для каждого пола добавляем группу SVG во

внутреннюю диаграмму. Для каждой группы присваиваем атрибуту `stroke` цвет `slateGray` (#305252), объявленный в `shared-constants.js`, а атрибуту `stroke-width` значение константы `boxplotStrokeWidth`, объявленной ранее. Элементы, добавленные в эту группу, унаследуют эти атрибуты, поэтому нам не придется устанавливать их несколько раз.

После этого мы добавляем элемент прямоугольника в группу SVG. Его атрибуту `x` (местоположение верхнего левого угла по горизонтали) мы присваиваем значение, возвращаемое шкалой `xScale` для указанного пола. Поскольку `xScale` возвращает положение центра ящичной диаграммы, мы должны вычесть половину ширины диаграммы, чтобы получить атрибут `x`. Атрибуту `y` (местоположение верхнего левого угла по вертикали) мы присваиваем координату `y` третьего квартиля, которую получаем, передавая третье значение из массива `quartiles` в шкалу `yScale`. Ширина прямоугольника задается с помощью `boxplotWidth`. Его высота вычисляется как разность местоположения первого и третьего квартилей (потому что ось `y` направлена сверху вниз!). Координаты `y` квартилей также возвращаются шкалой `yScale`. Наконец, мы присваиваем атрибуту `fill` прямоугольника значение `"transparent"`.

Медиана на диаграмме обозначается линией SVG. Линия пересекает прямоугольник по горизонтали слева направо. Горизонтальные координаты концов линии (`x1` и `x2`) вычисляются путем нахождения центра прямоугольника (`xScale(gender)`) и прибавлением или вычитанием половины ширины диаграммы. Вертикальные координаты (`y1` и `y2`) определяются путем передачи второго квартиля каждого пола в `yScale`. Чтобы визуально выделить медиану, мы задаем разный цвет ее рамки для женщин и мужчин (#826C7F и #FA7E61 соответственно, хранящиеся в константах `womenColor` и `menColor`, которые объявлены в `shared-constants.js`) и толщину рамки равной 10 пикселям.

Листинг 6.8 Рисование прямоугольников и медиан в ящичной диаграмме (box-plot.js)

```
const boxplotWidth = 60;
const boxplotStrokeWidth = 4;
```

Объявление констант для хранения значений ширины ящичной диаграммы и толщины рисок

```
genders.forEach(gender => {
```

Цикл по массиву полов

```
  const boxplotContainer = innerChart
    .append("g")
      .attr("stroke", slateGray)
      .attr("stroke-width", boxplotStrokeWidth);
```

Добавить группу SVG во внутреннюю диаграмму. Задать ее атрибуты `stroke` и `stroke-width`, которые будут унаследованы дочерними элементами

```
  boxplotContainer
    .append("rect")
      .attr("x", xScale(gender) - boxplotWidth/2)
      .attr("y", gender === "Female"
        ? yScale(femalesQuartiles[2])
        : yScale(malesQuartiles[2]))
      .attr("width", boxplotWidth)
      .attr("height", gender === "Female"
        ? yScale(femalesQuartiles[0]) -
            yScale(femalesQuartiles[2])
        : yScale(malesQuartiles[0]) -
            yScale(malesQuartiles[2]))
      .attr("fill", "transparent");
```

Добавить элемент прямоугольника в группу. Рассчитать его местоположение (атрибуты `x`, `y`, `width` и `height`), используя объявленные ранее шкалы. Присвоить атрибуту `fill` значение `"transparent"`

```
  boxplotContainer
    .append("line")
      .attr("x1", xScale(gender) - boxplotWidth/2)
      .attr("x2", xScale(gender) + boxplotWidth/2)
      .attr("y1", gender === "Female"
        ? yScale(femalesQuartiles[1])
        : yScale(malesQuartiles[1]))
      .attr("y2", gender === "Female"
        ? yScale(femalesQuartiles[1])
        : yScale(malesQuartiles[1]))
      .attr("stroke", gender === "Female"
        ? womenColor
        : menColor)
      .attr("stroke-width", 10);
```

Добавить горизонтальную линию, обозначающую медиану. Задать ее атрибуты `x1`, `x2`, `y1` и `y2`

```
});
```

В листинге 6.9 мы добавляем усы сверху и снизу, рисуя линии SVG. Нижний ус – это вертикальная линия, идущая от минимального значения заработной платы до первого квартиля, и горизонтальная линия, пересекающая ее в позиции, соответствующей минимальной заработной плате. Верхний ус – это вертикальная линия от третьего квартиля до максимальной заработной платы и горизонтальная линия, пересекающая ее в позиции, соответствующей максимальной заработной плате.

Листинг 6.9 Рисование усов ящичной диаграммы (box-plot.js)

```
genders.forEach(gender => {

  ...

  boxplotContainer
    .append("line")
      .attr("x1", xScale(gender))
      .attr("x2", xScale(gender))
      .attr("y1", gender === "Female"
```

Добавить вертикальную линию нижнего уса от минимальной зарплаты до первого квартиля

```
        ? yScale(femalesExtent[1])
        : yScale(malesExtent[1]))
      .attr("y2", gender === "Female"
        ? yScale(femalesQuartiles[2])
        : yScale(malesQuartiles[2]));

  boxplotContainer
    .append("line")
      .attr("x1", xScale(gender) - boxplotWidth/2)
      .attr("x2", xScale(gender) + boxplotWidth/2)
      .attr("y1", gender === "Female"
        ? yScale(femalesExtent[0])
        : yScale(malesExtent[0]))
      .attr("y2", gender === "Female"
        ? yScale(femalesExtent[0])
        : yScale(malesExtent[0]));

  boxplotContainer
    .append("line")
      .attr("x1", xScale(gender))
      .attr("x2", xScale(gender))
      .attr("y1", gender === "Female"
        ? yScale(femalesQuartiles[0])
        : yScale(malesQuartiles[0]))
      .attr("y2", gender === "Female"
        ? yScale(femalesExtent[0])
        : yScale(malesExtent[0]));

  boxplotContainer
    .append("line")
      .attr("x1", xScale(gender) - boxplotWidth/2)
      .attr("x2", xScale(gender) + boxplotWidth/2)
      .attr("y1", gender === "Female"
        ? yScale(femalesExtent[1])
        : yScale(malesExtent[1]))
      .attr("y2", gender === "Female"
        ? yScale(femalesExtent[1])
        : yScale(malesExtent[1]));

});
```

Добавить вертикальную линию нижнего уса от минимальной зарплаты до первого квартиля

Добавить горизонтальную линию нижнего уса на уровне минимальной зарплаты

Добавить вертикальную линию верхнего уса от третьего квартиля до максимальной зарплаты

Добавить горизонтальную линию верхнего уса на уровне максимальной зарплаты

После этого ящичные диаграммы будут выглядеть примерно так, как на рис. 6.15. Обратите внимание, что диаграммы, получившиеся у вас, могут немного отличаться от показанных здесь, поскольку значения зарплат генерируются случайным образом в `load-data.js`.

Если вы все еще не уверены в ценности ящичных диаграмм, то взгляните на рис. 6.16. На нем дополнительно изображены фактические точки данных, визуализированные ящичными диаграммами. Вертикальное положение каждого кружка соответствует зарплате, которую он представляет, а горизонтальное положение выбрано

случайным образом, чтобы уменьшить перекрытие. Обратите внимание, что плотность кружков выше внутри прямоугольников, где сосредоточено 50 % точек данных, и эта плотность уменьшается по мере удаления от прямоугольников вдоль усов.

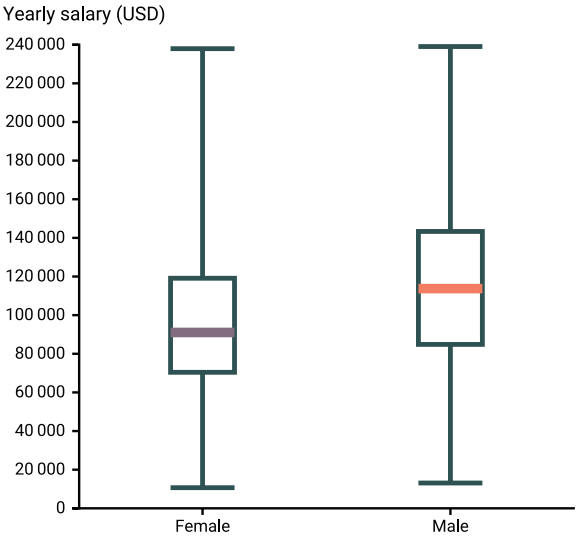


Рис. 6.15 Диаграммы распределения заработной платы женщин и мужчин, работающих в сфере визуализации данных в США

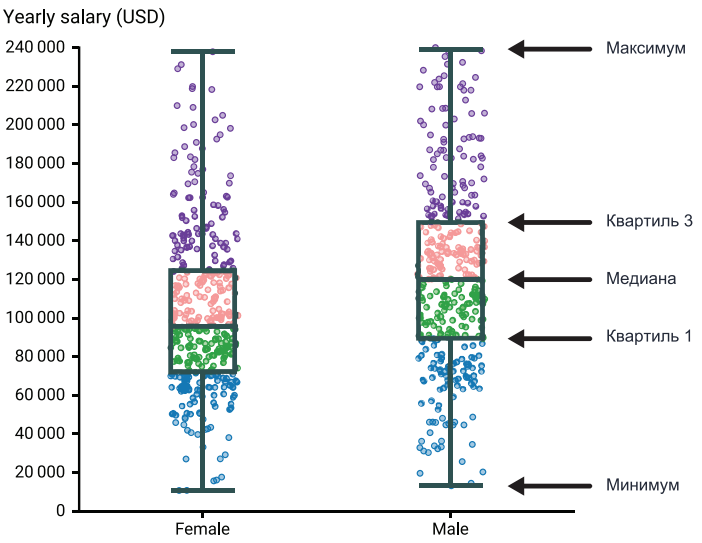


Рис. 6.16 Точки данных, представляющие зарплаты и визуализированные с помощью ящичных диаграмм. Вертикальное положение каждого кружка соответствует размеру зарплаты, которую он представляет, а горизонтальное положение рассчитывается случайным образом, чтобы уменьшить перекрытие

6.5 Сравнение распределений с помощью скрипичных диаграмм

В завершение этой главы мы познакомимся с визуализацией, объединяющей многие из понятий, обсуждавшихся выше: скрипичной диаграммой. Скрипичная диаграмма изображается как зеркальные изогнутые области, расширяющиеся и сужающиеся в зависимости от количества точек данных в разных областях. Они часто встречаются в медицинских диаграммах, описывающих дозировку и эффективность, но также используются в более общих случаях для визуализации распределений. В отличие от ящичных диаграмм, отображающих только предопределенную информацию (минимум, максимум, медиану и квартили), скрипичные диаграммы представляют все распределение. Они часто объединяются с дополнительной информацией, такой как положение среднего значения и квартилей.

В этом разделе мы сравним распределение зарплат пяти ведущих профессий в сфере визуализации данных: аналитика (analyst), разработчика (developer), дизайнера (designer), специалиста по данным (scientist) и руководителя (leadership). Мы решили не учитывать другие профессии, такие как картограф и преподаватель, из-за малого количества данных по ним. На рис. 6.17 показаны скрипичные диаграммы, которые мы построим для каждой из пяти профессий. Как видите, они объединены линией вдоль межквартильного размаха и точкой в позиции средней зарплаты. Мы также могли бы выбрать медиану вместо среднего значения, поскольку оба подхода допустимы.

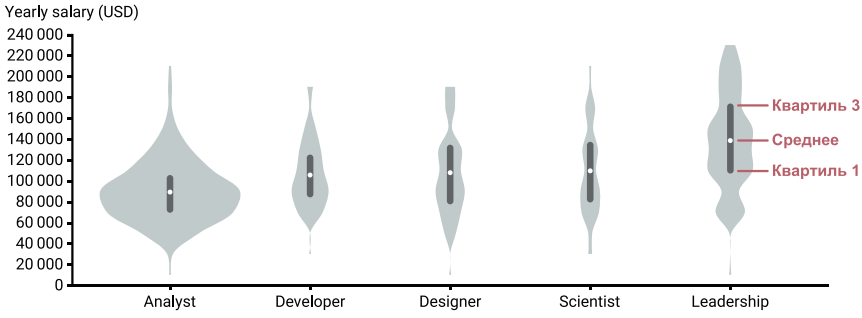


Рис. 6.17 Скрипичные диаграммы распределения зарплат пяти ведущих профессий в сфере визуализации данных. Каждая диаграмма включает серую линию, проходящую от первого до третьего квартиля, и белую точку, обозначающую среднюю зарплату

Откройте файл `violins.js` и найдите функцию `drawViolinCharts()` – мы начнем работать здесь. Обратите внимание, что файл

уже содержит код, добавляющий в DOM контейнер SVG и внутреннюю диаграмму, а также объявление массива с именем `roles`, включающий список объектов с идентификаторами профессий, которые мы будем визуализировать.

Чтобы построить скрипичные диаграммы, нам нужно отобрать зарплаты для каждой профессии, поместить их в отдельные корзины и вычислить их средние значения и квартили. Код в листинге 6.10 выполняет все эти вычисления, производя цикл по массиву `roles`. Для выбора зарплат, соответствующих текущей профессии, используется JavaScript-метод `filter()`, в цепочку к которому добавляется метод `map()`, генерирующий массив, содержащий только значения зарплат. Этот массив сохраняется в атрибуте `salaries`, чтобы сделать его доступным в массиве `roles`.

Затем с помощью метода `d3.mean()` вычисляется среднее значение по массиву `salaries`. Этот метод находится в модуле d3-array и работает подобно методам `d3.min()` и `d3.max()`.

Зарплаты организуются в корзины с помощью `d3.bin()`, а вычисление квартилей возлагается на квантильную шкалу, как было описано выше в этой главе. Все эти значения тоже сохраняются в массиве `roles`.

Наконец, мы сортируем профессии в порядке возрастания средних значений, используя JavaScript-метод `sort()`. Это улучшит читаемость диаграммы и облегчит сравнение.

Листинг 6.10 Извлечение информации для скрипичных диаграмм (violins.js)

```
const roles = [
  {id: "Designer" },
  {id: "Scientist" },
  {id: "Developer" },
  {id: "Analyst" },
  {id: "Leadership" },
];

roles.forEach(role => {

  role["salaries"] = data
    .filter(d => d.role === role.id)
    .map(d => d.salary);

  role["mean"] = d3.mean(role.salaries);

  role["bins"] = d3.bin()(role.salaries);
```

Массив профессий

Цикл по массиву профессий. Отобразить зарплаты для текущей профессии в массив. Сохранить полученный массив под ключом "salaries"

Вычислить средние зарплаты. Сохранить под ключом "mean"

Разбить массив зарплат на корзины d3.bin(). Сохранить корзины под ключом "bins"

```
  const quartilesScale = d3.scaleQuantile()
    .domain(role.salaries)
    .range([0, 1, 2, 3]);
  role["quartiles"] = quartilesScale.quantiles();
});

roles.sort((a, b) => a.mean - b.mean);
```

Вычислить квартили массива зарплат с помощью квантильной шкалы. Сохранить квартили под ключом "quartiles"

Отсортировать профессии в порядке возрастания средних зарплат

На рис. 6.18 показано, что для построения скрипичных диаграмм нам понадобятся три шкалы:

- точечная шкала для распределения профессий по оси `x`;
- линейная шкала значений заработной платы по оси `y`;
- линейная шкала для расчета ширины скрипичных диаграмм на основе количества точек данных в каждой корзине.

В листинге 6.11 мы объявляем точечную шкалу, отвечающую за распределение профессий по оси `x`, и сохраняем ее в константе с именем `xScale`. Как обсуждалось в разделе 6.4.2, `d3.scalePoint()` принимает дискретную предметную область – в данном случае массив профессий, созданный с помощью JavaScript-метода `map()`, – и возвращает значения из непрерывного диапазона – пространства, определяемого шириной внутренней диаграммы. Затем задаем отступы на двух концах оси, вызывая метод `padding()` и передавая ему значение `0.7` – 70 % от размера шага шкалы.

Далее объявляем константу `yScale` – линейную шкалу для распределения зарплат по оси `y`. Предметная область шкалы непрерывна и простирается от 0 до максимальной заработной платы в наборе данных. Диапазон тоже непрерывен и включает значения вдоль вертикальной оси внутренней диаграммы. В цепочку создания этой шкалы мы добавляем метод `nice()`, чтобы гарантировать, что ось `y` заканчивается круглым числом.

Наконец, мы объявляем еще одну линейную шкалу, `violinsScale`, которая будет отвечать за вычисление ширины скрипичных диаграмм. Эта ширина будет меняться вдоль оси `y` в зависимости от количества точек данных в каждом диапазоне зарплат.

Листинг 6.11 Объявление шкал для скрипичных диаграмм (violins.js)

```
const xScale = d3.scalePoint()
  .domain(roles.map(d => d.id))
  .range([0, innerWidth])
  .padding(0.7);
```

Объявление точечной шкалы, отвечающей за распределение профессий вдоль оси `x`

```
const maxSalary = d3.max(data, d => d.salary);
const yScale = d3.scaleLinear()
  .domain([0, maxSalary])
  .range([innerHeight, 0])
  .nice();
```

Объявление линейной шкалы для распределения зарплат вдоль оси `y`

```
let maxBinLength = 0;
roles.forEach(role => {
  const max = d3.max(role.bins, d => d.length);
  if (max > maxBinLength) {
    maxBinLength = max;
  }
});
```

Определить размер самой большой корзины среди профессий

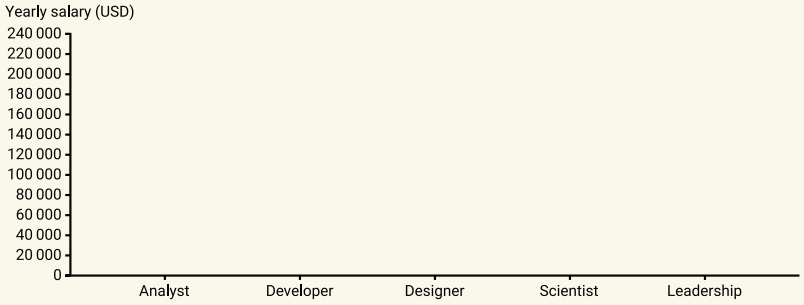
```
const violinsScale = d3.scaleLinear()
  .domain([0, maxBinLength])
  .range([0, xScale.step()/2]);
```

Объявление линейной шкалы, отвечающей за вычисление ширины скрипичных диаграмм для каждого диапазона зарплат

Упражнение: добавьте оси

В предыдущих главах вы уже видели, как добавить оси в диаграмму. Теперь настал момент сделать это самостоятельно! Используйте шкалы, объявленные в листинге 6.11, и нарисуйте оси для наших скрипичных диаграмм. Профессии должны отображаться вдоль оси x, а зарплаты – вдоль оси y. Также добавьте метку к оси y с текстом «Yearly salary (USD)» (Годовая зарплата (USD)).

По завершении оси должны выглядеть примерно так, как показано на рисунке ниже. Если у вас возникнут сложности или вы захотите сравнить свое решение с нашим, то загляните в листинг D.3 в приложении D или в файл `6.5-Violins/end` в репозитории с примерами для этой главы.



Оси для скрипичных диаграмм. Профессии расположены вдоль оси x, а зарплаты – вдоль оси y

Скрипичные диаграммы можно рассматривать как сглаженные гистограммы, зеркально отраженные относительно невидимой центральной оси. Для иллюстрации этой идеи создадим сначала вертикально ориентированные гистограммы для каждой профессии.

Код в листинге 6.12 перебирает в цикле профессии и для каждой добавляет группу SVG во внутреннюю диаграмму, которая сохраняется в константе с именем `roleContainer`. Такой подход поможет нам сохранить разметку аккуратной и простой для проверки.

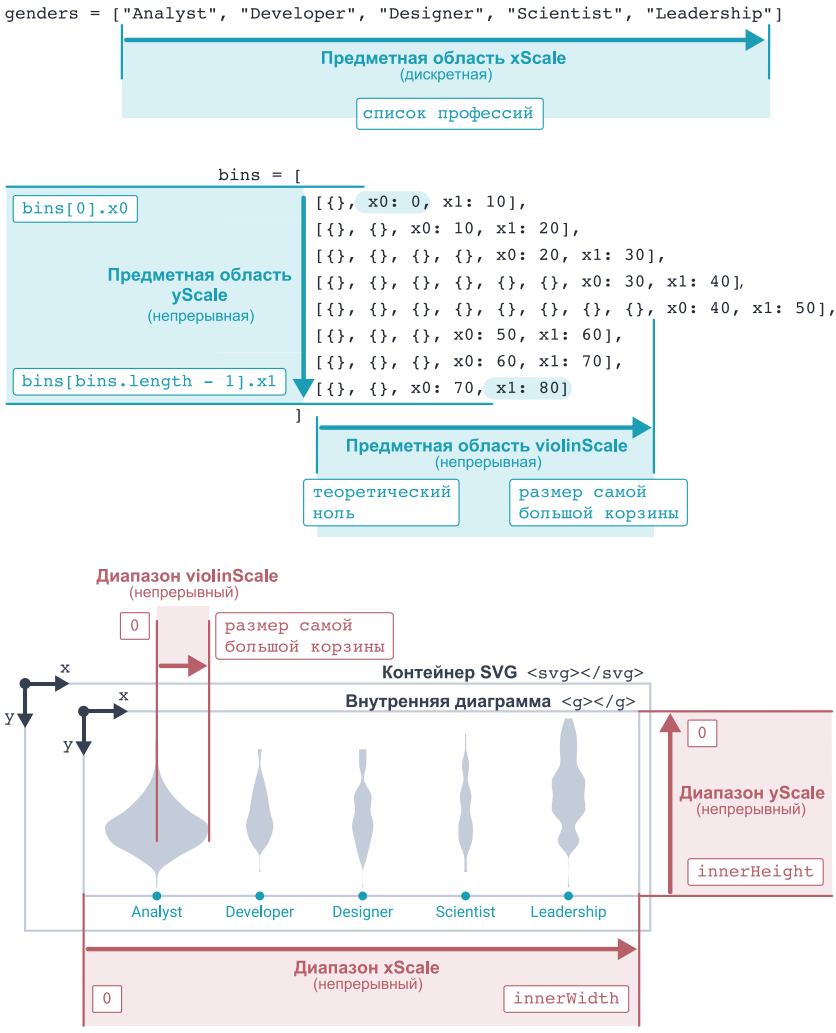


Рис. 6.18 Для построения скрипичных диаграмм нужны три шкалы: точечная шкала, отвечающая за распределение профессий по оси x, линейная шкала, отвечающая за распределение значений зарплат по оси y, и линейная шкала для расчета ширины скрипичных диаграмм на основе количества точек данных в каждой корзине

Затем используется шаблон привязки данных, чтобы добавить элемент прямоугольника для каждой корзины в пределах профессии. Прямоугольники размещаются вдоль центральной оси текущей роли, горизонтальная позиция которой возвращается шкалой `xScale`. Их ширина пропорциональна количеству точек данных в соответствующих корзинах и вычисляется с помощью `violinScale`. Наконец, вертикальная позиция и высота прямоугольников зависят от диапазона зарплаты и возвращаются шкалой `yScale`.

Листинг 6.12 Рисование гистограммы для каждой роли (violins.js)

```
roles.forEach(role => {

  const roleContainer = innerChart
    .append("g");

  roleContainer
    .selectAll(`.bar-${role.id}`)
    .data(role.bins)
    .join("rect")
      .attr("class", `bar-${role.id}`)
      .attr("x", xScale(role.id))
      .attr("y", d => yScale(d.x1))
      .attr("width", d => violinsScale(d.length))
      .attr("height", d => yScale(d.x0) - yScale(d.x1))
      .attr("fill", slateGray)
      .attr("fill-opacity", 0.4)
      .attr("stroke", white)
      .attr("stroke-width", 2);

});
```

Цикл по профессиям

Добавить группу SVG во внутреннюю диаграмму. Сохранить ее в константе с roleContainer

Использовать шаблон привязки данных, чтобы добавить прямоугольник для каждой корзины текущей роли. Прямоугольники размещаются вдоль центральной оси текущей профессии, возвращаемой шкалой xScale. Их ширина пропорциональна количеству точек данных внутри корзин и вычисляется с помощью violinScale. Вертикальные позиции прямоугольников зависят от диапазона зарплаты и вычисляются с помощью yScale

Теперь гистограммы должны выглядеть как на рис. 6.19. Рисование гистограмм не является обязательным шагом при создании скрипичных диаграмм, однако этот этап поможет вам понять, как вычислять значения для атрибута `d` элемента `path` скрипичной диаграммы.

Чтобы получить скрипичную диаграмму, достаточно нарисовать кривую, проходящую через конец каждого столбца гистограммы. В листинге 6.13 мы все еще продолжаем работать в цикле, перебирающем профессии. Начнем с объявления генератора области размаха и настройки его функций доступа. Нам нужно, чтобы область имела две границы по горизонтали. Первая (`x0`) расположена на центральной оси текущей профессии, а вторая (`x1`) – на конце соответствующего столбца. По вертикали точки `x1` располагаются посередине каждого столбца. Наконец, кривая сглаживается с помощью `d3.curveCatmullRom`. Более подробно о генераторах областей размаха рассказывается в главе 4.

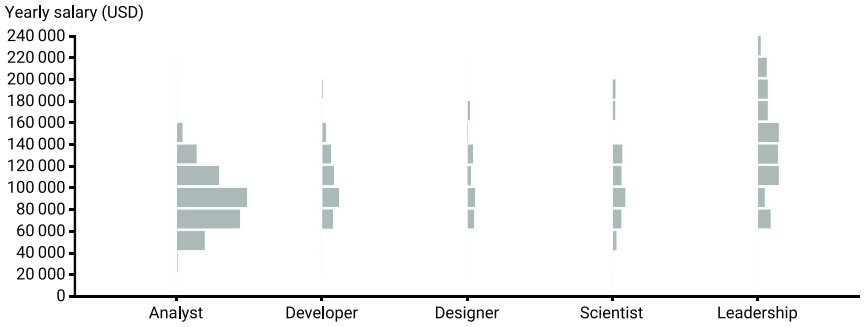


Рис. 6.19 Гистограммы распределения зарплат для каждой профессии

Затем мы добавляем элемент `path` в `roleContainer`. Чтобы вычислить его атрибут `d`, вызывается генератор области размаха, которому передаются корзины текущей профессии.

Листинг 6.13 Рисование половинок скрипичных диаграмм (violins.js)

```
roles.forEach(role => {
  ...

  const areaGenerator = d3.area()
    .x0(d => xScale(role.id))
    .x1(d => xScale(role.id) + violinsScale(d.length))
    .y(d => yScale(d.x1) + ((yScale(d.x0) -
      yScale(d.x1))/2))
    .curve(d3.curveCatmullRom);

  roleContainer
    .append("path")
      .attr("d", areaGenerator(role.bins))
      .attr("fill", "transparent")
      .attr("stroke", slateGray)
      .attr("stroke-width", 2);

};
```

Объявление генератора области размаха. Области имеют две границы по горизонтали, первая (`x0`) расположена на центральной линии каждой роли, а вторая (`x1`) – на концах столбцов гистограммы. По вертикали точки `x1` располагаются посередине каждого столбца

Добавить элемент `path` в `roleContainer`. Чтобы вычислить его атрибут `d`, вызывается генератор области размаха, которому передаются корзины текущей профессии

Теперь должны появиться диаграммы в форме половинки скрипки, как показано на рис. 6.20. Области размаха начинаются на центральной линии каждой профессии и простираются до концов столбцов гистограммы.

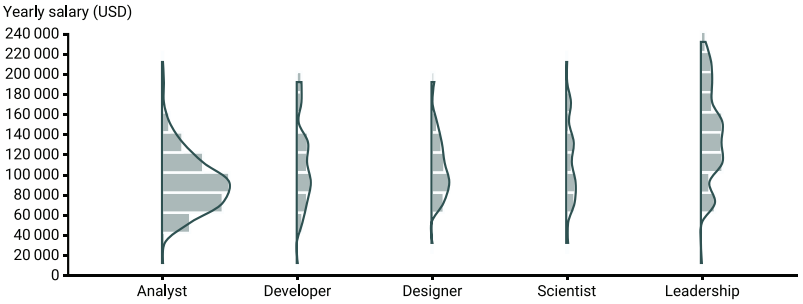


Рис. 6.20 Диаграммы в форме половинок скрипок, построенные добавлением кривой, проходящей через концы столбцов гистограммы

Чтобы завершить создание скрипичных диаграмм, закомментируем строки кода, создающие гистограмму. В JavaScript однострочные комментарии начинаются с `//`, а многострочные начинаются с пары символов `/*` и заканчиваются парой символов `*/`.

Наконец, чтобы придать скрипичным диаграммам законченный вид, нужно просто отразить правые половинки. В листинге 6.14 мы изменили функцию доступа `x0()` генератора области размаха, чтобы она отражала `x1()`. Затем удалили рамку, оконтуривающую элемент `path`, установили цвет заливки `slateGray`, доступный в `shared-constants.js` (#305252), и степень непрозрачности равной 30 %.

Листинг 6.14 Рисование полных скрипичных диаграмм (violins.js)

```
roles.forEach(role => {

  ...

  const areaGenerator = d3.area()
    .x0(d => xScale(role.id) - violinsScale(d.length))
    .x1(d => xScale(role.id) + violinsScale(d.length))
    .y(d => yScale(d.x1) + ((yScale(d.x0) - yScale(d.x1))/2))
    .curve(d3.curveCatmullRom);

  roleContainer
    .append("path")
      .attr("d", areaGenerator(role.bins))
      .attr("fill", slateGray)
      .attr("fill-opacity", 0.3);

};
```

Отразить границу x1

Удалить рамку вокруг области размаха, установить цвет slateGray заливки и задать непрозрачность равной 30 %

Как показано на рис. 6.21, у нас получились замечательные скрипичные диаграммы! Осталось добавить лишь несколько деталей, чтобы завершить проект этой главы, что вы и сделаете в следующем упражнении.

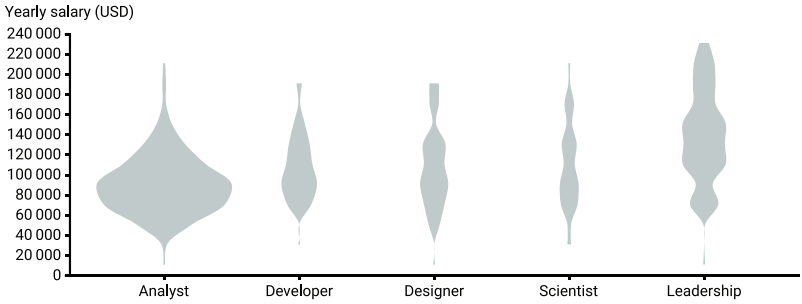


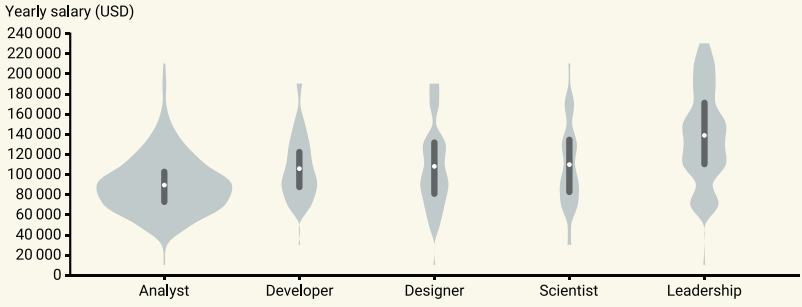
Рис. 6.21 Скрипичные диаграммы для каждой профессии

Упражнение: добавьте межквартильные диапазоны и средние значения

Для завершения проекта этой главы отобразите на скрипичных диаграммах межквартильные диапазоны и средние значения, как показано на следующем рисунке. В этом вам помогут следующие указания:

- межквартильный диапазон представлен прямоугольниками, простирающимися от первого до третьего квартиля. Эти значения доступны в массиве `roles`;
- прямоугольники имеют ширину 8 пикселей, а их углы имеют закругления с радиусом 4 пикселя. Цвет прямоугольников – серый, он задан в переменной `gray`, доступной в `shared-constants.js` (#606464);
- средние значения изображаются кружками;
- кружки имеют радиус 3 пикселя и белый цвет, который задан в переменной `white`, доступной в `shared-constants.js` (#faffff).

Если у вас возникнут сложности или вы захотите сравнить свое решение с нашим, то вы найдете его в листинге D.4 и в папке `6.5-Violins/end` с примерами кода для этой главы. Но мы призываем вас постараться выполнить данное упражнение самостоятельно, потому что практика – лучший способ научиться чему-либо! Обратите внимание, что ваше решение может немного отличаться от нашего, потому что в разработке часто один и тот же результат можно получить разными путями!



Скрипичные диаграммы с межквартильными диапазонами и средними значениями

На этом мы завершаем первый раздел книги! Теперь у вас должно сложиться хорошее представление об основных возможностях D3. В следующих главах мы погрузимся в более сложные вопросы, связанные с визуализацией.

Интервью с Морицем Стефанером и Фабианом Эхмелем

Стефанер – независимый дизайнер и консультант по визуализации данных.
Эхмель – технолог-дизайнер в Studio NAND.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

Фабиан: Сначала я изучал географию и информатику, а затем продолжил изучать приемы организации интерактивных взаимодействий. Это сочетание предметов привело меня прямиком к визуализации данных, и во время учебы я участвовал в создании нескольких таких проектов. Перед окончанием университета я несколько месяцев стажировался в Accurat в Милане, занимаясь проектированием визуализаций данных. После окончания я поступил на работу в Studio NAND в Берлине, где занимался проектами визуализации данных, такими как Impfdashboard.

Мориц: До изучения когнитивистики и проектирования интерфейсов в начале 2000-х я работал в агентствах веб-дизайна. С 2007 года я основал свое индивидуальное предприятие «Truth & Beauty Operator» и помогал клиентам из промышленных и академических кругов находить правдивые и красивые представления данных. Я создал несколько экспериментальных проектов – от скульптур данных и интерактивных инсталляций до использования еды в качестве носителя, – чтобы изучить, насколько далеко можно зайти в области визуализации данных. В последнее время я в основном работаю над крупными системами проектирования и экосистемами для инструментов визуализации данных.

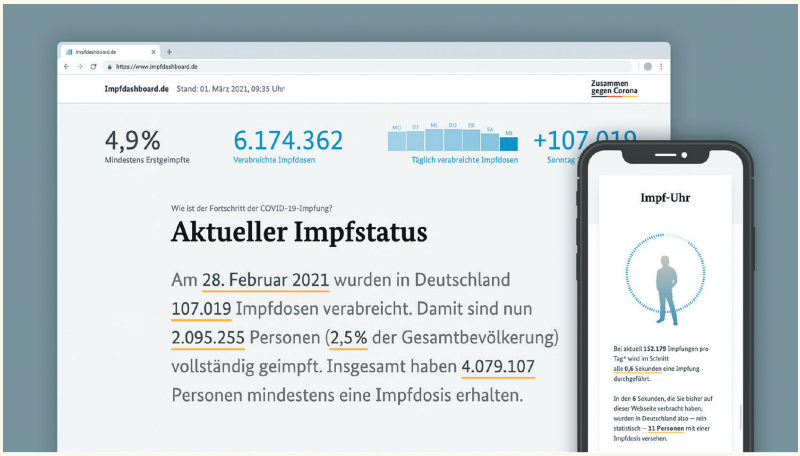
Какими инструментами визуализации данных вы в основном пользуетесь или предпочитаете? Какое место в вашем рабочем процессе занимает D3?

Фабиан: В Studio NAND подавляющее большинство проектов основано на веб-окружении, поэтому для их разработки обычно используется какой-то из популярных веб-фреймворков: React или Svelte. В обоих случаях для подготовки данных и их визуализации мы используем D3. В основном мы применяем эту библиотеку для создания шкал, анализа данных, форматирования дат и времени, создания макетов с силовой компоновкой, проецирования геоданных, интерполяции кривых и отображения осей диаграмм. Но мы не используем возможности D3 для изменения DOM, полностью полагаясь на логику React и Svelte.

Мориц: Как и Фабиану, мне нравится использовать Svelte и D3. В последнее время мне особенно нравится модуль `plot.js` из Observable. Он предлагает гораздо более простой и лаконичный способ постройки

стандартных диаграмм, чем довольно низкоуровневый подход построения каждого элемента по отдельности.

Представленный проект: «Covid-19 Vaccination Dashboard – Impfdashboard» (https://impfdashboard.de/en/), совместный проект Cosmonauts & Kings GmbH (https://cosmonautsandkings.com), Studio NAND GmbH (https://nand.io/) и Морица Стефанера (Moritz Stefaner) (https://truth-and-beauty.net/)



Impfdashboard – официальный немецкий дашборд с информацией о вакцинации от COVID-19, разработанный по заказу Федерального министерства здравоохранения Германии (источник: Cosmonauts & Kings GmbH совместно со Studio NAND GmbH и Truth & Beauty)

Impfdashboard – официальный немецкий дашборд с информацией о вакцинации от COVID-19, разработанный по заказу Федерального министерства здравоохранения Германии. Мы спроектировали и разработали его в сотрудничестве с Cosmonauts & Kings.

Дашборд предназначался для широких масс и специально создавался так, чтобы представляемая им информация была понятна широкому кругу посетителей. Поэтому мы представили несколько важных показателей (таких как количество введенных доз вакцины, ежедневный уровень вакцинации и процент от общего числа вакцинированных) в визуальной и письменной форме.

Для посетителей, заинтересованных в более подробной информации, Impfdashboard предоставляет несколько дополнительных визуализаций. Они сообщают о ситуации в разных регионах и возрастных группах, а также о течении кампании по вакцинации.

Расскажите, как создавался этот проект.

Из-за пандемии мы столкнулись с некоторыми ограничениями при разработке. Первоначальную версию дашборда нужно было спроектиро-

вать и разработать всего за несколько недель. Поэтому мы использовали очень гибкий подход, начав с определения информационных потребностей посетителей и коммуникативных целей, создания макетов и начального языка визуального дизайна. Мы продолжали применять этот подход в течение следующих двух с половиной лет, создавая прототипы новых визуализаций с помощью Svelte и D3.

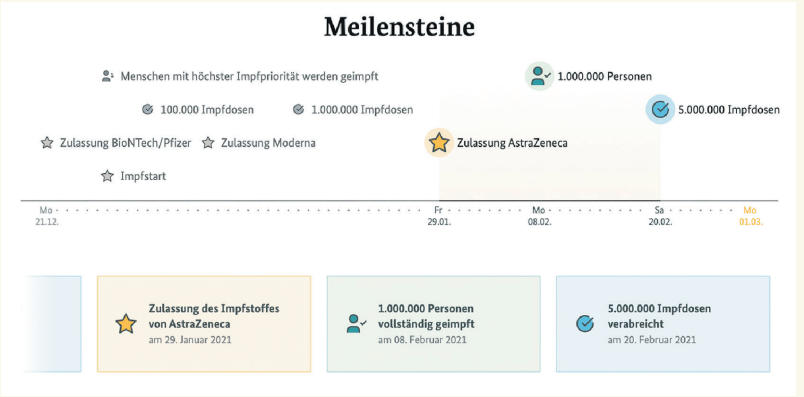
Данные для дашборда предоставлены Институтом Роберта Коха (Robert Koch Institute, RKI) – центральным научным учреждением правительства Германии по биомедицине и общественному здравоохранению. Первое время нам приходилось выпускать обновления данных на сайте вручную каждый день, опираясь на таблицы Excel. Позже мы усовершенствовали этот процесс и использовали конвейер данных для автоматического обновления сайта всякий раз, когда появлялись новые данные.

По ходу поддержки проекта в течение длительного периода нам повезло получить большое количество отзывов пользователей. Мы активно реагировали на них, стараясь улучшить существующие или будущие особенности.

Этот проект – в первую очередь мобильный. Как вы справились с трудностями создания дашборда, который хорошо выглядит и на маленьких сенсорных экранах, и на больших мониторах настольных компьютеров?

Для нас было важно сделать Impfdashboard максимально доступным. Поэтому мы выбрали дизайн, ориентированный на мобильные устройства, и сосредоточились на доступности, удобстве использования и простоте представления информации.

Как уже говорилось, сайт предоставляет самую актуальную ежедневную информацию, поэтому посетителям не нужно прокручивать страницу, чтобы получить ежедневную дозу сведений. Кроме того, мы упростили некоторые диаграммы в мобильном представлении, чтобы сделать их более удобными для чтения и понимания. Мы также сделали компоненты пользовательского интерфейса адаптируемыми к устройству. Один из примеров – ползунок, обеспечивающий плавную навигацию внутри диаграммы на мобильном устройстве без использования сложных сенсорных жестов.



Текст и значки используются для передачи информации и улучшения доступности (источник: Cosmonauts & Kings GmbH совместно со Studio NAND GmbH и Truth & Beauty)

Вы стремились «упростить» представление информации, используя для этого текст и значки. Мориц, на своем сайте вы упоминаете, что в визуализациях текст играет вспомогательную роль. Это звучит очевидно, но этим часто пренебрегают. Расскажите чуть больше об этом.

Конечно! В последнее время я все больше стал размышлять о доступности интересоваться ролью текста в визуализации данных. Так же, как и диаграммы, текст может автоматически генерироваться и использоваться для передачи информации. По сравнению с визуальной диаграммой, текст, сгенерированный из данных, потенциально более понятен, удобен для чтения с экрана, легко копируется и может автоматически переводиться на разные языки. Поэтому я думаю, что текст – идеальное дополнение к диаграммам и большему количеству визуальных элементов.

Планируете ли вы снова сотрудничать в будущем? Над какими проектами вы хотели бы работать?

Impfdashboard – это не первый совместный проект Studio NAND и Морица. Нам уже приходилось работать над другими проектами визуализации данных, такими как Peak Spotting, разработанный для немецкой железнодорожной компании Deutsche Bahn. Мы с удовольствием продолжим сотрудничество, так как наши навыки прекрасно дополняют друг друга.

Если проект окажется недоступным по указанному выше адресу, то его можно найти в веб-архиве (http://mng.bz/1GzR).

Итоги

- Для визуализации распределений часто требуется предварительная обработка набора данных путем группировки в корзины – группы одинаковой ширины вдоль оси. В D3 это делается с помощью метода `d3.bin()`.
- Для рисования гистограмм используется шаблон привязки данных, с помощью которого производится добавление прямоугольников на основе полученных корзин с данными. Длина каждого прямоугольника пропорциональна количеству точек данных в соответствующей корзине.
- Пирамидальная диаграмма похожа на гистограмму, за исключением того, что используется для сравнения двух распределений.
- Ящичные диаграммы состоят из прямоугольника, простирающегося от первого до третьего квартиля, и усов, простирающихся от минимального значения до первого квартиля и от третьего квартиля до максимального значения. Часто дополнительно отображается медиана в виде перпендикулярной линии.
- В D3 квартили вычисляются с помощью квантильной шкалы `d3.scaleQuantile()`, которая принимает дискретную предметную область домена и дискретный диапазон. Чтобы получить квартили и разделить данные на четыре группы, диапазон должен быть представлен массивом из четырех элементов, например `[0, 1, 2, 3]`.
- Чтобы нарисовать ящичную диаграмму с помощью D3, нужно просто добавить прямоугольные и линейные элементы в выборку и задать их атрибуты на основе положений минимального, максимального, медианного и квартильного значений.
- Точечная шкала `d3.scalePoint()` похожа на интервальную шкалу, обсуждаемую в главе 3, но имеет нулевую ширину каждого интервала. Ее предметная область состоит из дискретного списка элементов, распределенных по непрерывному диапазону.
- Скрипичные диаграммы – это зеркальные, изогнутые области, выпирающие в местах, где много точек данных, и сужающиеся там, где их мало. Их можно описать как сглаженные гистограммы, зеркально отраженные вокруг невидимой центральной оси. Чтобы нарисовать скрипичную диаграмму, нужно сгенерировать область размаха, проходящую через кончики каждого прямоугольника. Часто полезно сначала нарисовать гистограммы, чтобы определить, как действовать дальше.
- Скрипичные диаграммы нередко комбинируют с прямоугольником, показывающим межквартильный диапазон, и кружком, расположенным в точке среднего или медианного значения.

Часть II

Новые стандарты

С появления D3 в 2011 году многое изменилось! В настоящее время более половины цифрового контента потребляется пользователями смартфонов. Многие также ожидают, что цифровой контент будет интерактивным и даже анимированным. Набор инструментов разработчиков тоже эволюционировал, и большинство современных проектов создаются с использованием JavaScript-фреймворков, таких как React или Svelte. Мы также наконец пришли к пониманию, что веб-контент не всем одинаково доступен и мы (мучительно) медленно адаптируем наши привычки разработки.

В этой части книги мы рассмотрим новые стандарты, которым должны следовать, создавая цифровые визуализации данных. В главе 7 мы посмотрим, как создавать интерактивные взаимодействия и анимацию с помощью D3. В главе 8 обсудим три стратегии объединения D3 с фреймворком. В главе 9 вернемся к визуализациям, созданным ранее, и сделаем их интерактивными. А в главе 10 сделаем цифровые визуализации более доступными для людей с ограниченными возможностями.

7 Интерактивные визуализации

В этой главе рассматриваются:

- добавление обработчиков событий в выборку D3;
- реализация плавных переходов;
- фильтрация содержимого визуализации;
- использование всплывающих подсказок для отображения дополнительной информации;
- анимация появления, обновления и исчезновения элементов визуализации.

Одно из ключевых преимуществ D3 – простота создания интерактивных визуализаций, соответствующих современным веб-стандартам. Это, пожалуй, одна из основных причин, почему так много специалистов по работе с данными хотят освоить эту библиотеку.

В первой части этой книги мы усердно знакомились с философией D3 и ее строительными блоками. Теперь самая пора развлечься! В этой главе мы используем ранее построенные диаграммы и сделаем их интерактивными. Для начала добавим фильтрацию по полу в гистограмму, созданную в главе 6. Благодаря этому упражнению вы узнаете, как получать и обрабатывать события пользователя и создавать плавные переходы. Затем мы вернемся к линейной диаграмме температуры, созданной в главе 4, и отобразим всплывающую подсказку при наведении указателя мыши на точку данных. Мы

продолжим развитие этой возможности в следующем разделе и добавим составные подсказки, показывающие распределение продаж каждого носителя музыки при перемещении указателя мыши вдоль потокового графика из главы 5. Наконец, мы создадим новую диаграмму рассеяния, чтобы показать вам, как D3 обеспечивает детальный контроль над переходами, когда данные появляются в визуализации или исчезают из нее. Мы также познакомимся с логарифмическими и радиальными шкалами D3.

Эта глава охватывает не все типы интерактивных взаимодействий, доступные в D3, тем не менее она поможет вам заложить прочную основу для создания интерактивных визуализаций в будущем. В оставшейся части книги мы исследуем некоторые другие взаимодействия, такие как окрашивание, масштабирование и панорамирование. Но сначала давайте кратко обсудим, зачем создавать интерактивные визуализации, и перечислим основные практики.

7.1 Зачем нужна интерактивность?

Интерактивные визуализации предлагают более широкий спектр возможностей по сравнению со статическими визуализациями. Они позволяют исследовать наборы данных, выделять связи или фокусироваться на подмножестве данных. Интерактивные взаимодействия также могут сделать информацию более доступной и удобной для поиска. Интерактивные визуализации не ограничиваются точкой зрения их создателя и возвращают власть пользователям. Их можно с успехом использовать для изучения проекта, поиска отклонений и выводов. Как говорит Энди Кирк (Andy Kirk), «[Интерактивные визуализации] расширяют физические границы доступности информации в данном пространстве»[1]. Интерактивность – безусловно, увлекательная тема. Но важно помнить, что не каждая визуализация может или должна быть интерактивной.

7.1.1 Некоторые лучшие практики интерактивности

При планировании взаимодействий важно помнить о конечном пользователе. Задавая себе следующие вопросы, вы увеличиваете шансы на создание взаимодействий, которые действительно необходимы:

- *Сколько времени пользователь готов потратить на изучение визуализации?* Ответ на этот вопрос зависит от контекста. Если визуализация является частью панели мониторинга, то критически важная информация должна быть доступна напрямую,

[1] Andy Kirk, *Data Visualisation: A Handbook for Data Drive Design*, 2nd ed. (London: SAGE Publications Ltd, 2019), 203.

а взаимодействия должны помогать получать ответы на дополнительные вопросы. Если, напротив, визуализация является частью длинной онлайн-статьи, то можно предположить, что у пользователя есть время на изучение деталей.

- *Где в основном будет отображаться визуализация, на настольных компьютерах или мобильных устройствах?* На настольных компьютерах можно использовать события мыши, а на мобильных устройствах – события сенсорных экранов. На сенсорных экранах нужно планировать больше места для сенсорных взаимодействий, потому что наши пальцы больше указателя мыши.
- *Насколько хорошо пользователь знает предмет?* Это помогает решить: отображать ли основную информацию напрямую или можно «скрыть» ее в интерактивных функциях.
- *Ориентируется ли пользователь в технологиях?* Поймет ли он, как взаимодействовать с визуализацией? Подумайте о предоставлении инструкций, когда это уместно.
- *Дают ли интерактивность и анимация какие-то преимущества?* Мы должны определить, улучшают ли они пользовательский опыт или они просто демонстрируют, насколько мы крутые.
- *Не мешает ли интерактивность извлечению информации из визуализации?* Если мешает, то подумайте, как можно изменить способ взаимодействия, или используйте прозрачность для уменьшения помех.

7.2 Фильтрация визуализации

Один из типичных вариантов интерактивности – фильтрация визуализации. В этом разделе мы продолжим работу над гистограммой, созданной в главе 6, и позволим пользователю фильтровать данные по полу. Откройте готовый проект, доступный по адресу http://mng.bz/n1na, понажимайте на кнопки над гистограммой и посмотрите, как происходят переходы. Как показано на рис. 7.1, у нас есть три кнопки, реализующие фильтрацию: одна (она выбрана по умолчанию) обеспечивает просмотр всех данных, одна служит для просмотра данных, полученных от респондентов, которые идентифицируют себя как женщины, и одна – от респондентов, которые идентифицируют себя как мужчины.

> **ПРИМЕЧАНИЕ** Загрузите или клонируйте файлы из репозитория книги на GitHub (http://mng.bz/Xqjv). Папка `chapter_07/7.2-Filtering` содержит две подпапки с именами `start` и `end`. Откройте папку `start` в редакторе и запустите локальный веб-сервер. Если у вас возникнут трудности, обратитесь к решению в папке `end`.

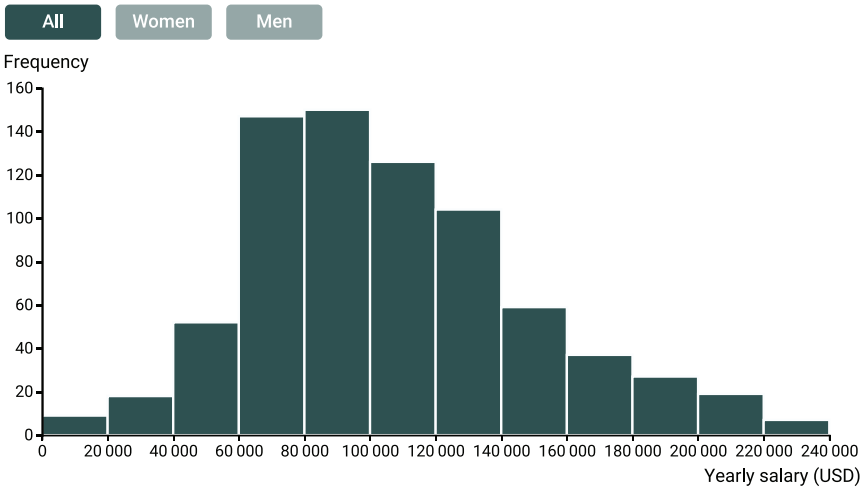


Рис. 7.1 Гистограмма распределения зарплат среди специалистов по визуализации данных в США с кнопками для фильтрации данных по полу

Файлы с кодом этого проекта уже содержат реализацию гистограммы, созданной в главе 6. Обратите внимание, что мы переместили объявление шкал в файл `shared-constants.js`, чтобы они были доступны глобально.

Сначала добавим кнопки фильтрации. Файл `shared-constants.js` уже содержит массив с именем `filters`, содержащий идентификатор, метку и статус для каждой кнопки. Этот массив можно увидеть в следующем фрагменте кода. Обратите внимание, что свойство `isActive` первого фильтра установлено в значение `true`, а остальные – в значение `false`. Мы будем использовать это свойство для определения выбранного фильтра:

```
const filters = [
  { id: "all", label: "All", isActive: true },
  { id: "female", label: "Women", isActive: false },
  { id: "male", label: "Men", isActive: false },
];
```

Чтобы добавить фильтры в интерфейс, откройте файл `interactions.js` и отыщите функцию `populateFilters()`. Эта функция уже вызывается из `load-data.js` и получает набор данных.

В листинге 7.1 мы сначала выбираем `div` с идентификатором `"filters"`, который уже существует в `index.html`. Затем используем шаблон привязки данных, чтобы добавить три кнопки в выборку, основываясь на трех объектах, содержащихся в массиве `filters`. Каждой кнопке мы присваиваем имя класса `"filter"`. Если свойство `isActive` фильтра имеет значение `true`, то мы также добавляем имя класса `"active"`. В завершение задаем текст, отображаемый на каждой кнопке, присваивая его свойству `label` фильтра.

Листинг 7.1 Добавление кнопок фильтров в интерфейс (interactions.js)

```
const populateFilters = (data) => {

  d3.select("#filters")
    .selectAll(".filter")
    .data(filters)
    .join("button")
      .attr("class", d => `filter ${d.isActive ?
        "active" : ""}`)
      .text(d => d.label);

};
```

Выбрать div с идентификатором "filters"

Использовать шаблон привязки данных, чтобы добавить кнопку для каждого объекта в массиве filters

Присвоить имя класса "filter" кнопке. Если свойство isActive фильтра равно true, то добавить имя класса "active"

Задать текст, отображаемый на кнопках, используя свойство label фильтра

Сохраните проект и откройте его в браузере. Вы увидите, что над гистограммой появились три кнопки, как показано на рис. 7.1, и оформлены они стилями CSS, перечисленными в `visualization.css`. Активная кнопка темно-зеленая, а остальные имеют полупрозрачный фон, как бы говоря, что они не выбраны.

Если вы понажимаете на кнопки, то ничего не произойдет, потому что мы еще не обрабатываем события нажатия. Как это сделать, мы расскажем в следующем разделе.

7.2.1 Обработка пользовательских событий

Чтобы гистограмма реагировала на нажатия кнопок, нужно прикрепить обработчики событий. *Обработчики событий* – это просто стратегии, которые используются для ожидания и обнаружения предопределенных событий. В JavaScript для этого используется метод `addEventListener()`. В D3 мы прикрепляем обработчики событий к выборке с помощью метода `on()`, который является частью модуля d3-selection.

Как показано на рис. 7.2, первый аргумент метода `on()` – это имя события, которое мы должны обрабатывать. Здесь можно использовать любое событие DOM, например `click`, `mouseover`, `touch`, `keydown` и т. д. Второй аргумент – это функция обратного вызова, выполняющая желаемое действие(я). Эта функция принимает два параметра: собственно событие и данные, прикрепленные к выборке D3, в которой произошло событие.

Прикрепим обработчик событий к нашим кнопкам. В следующем фрагменте мы добавили в цепочку вызов метода `on()`. Этому методу передаем тип события `"click"` в первом аргументе, соответственно, функция обратного вызова во втором аргументе `on()` будет вызываться каждый раз при нажатии кнопки:

```
d3.select("#filters")
  .selectAll(".filter")
  .data(filters)
  .join("button")
    .attr("class", d => `filter ${d.isActive ? "active" : ""}`)
    .text(d => d.label)
    .on("click", (e, d) => {
      console.log("DOM event", e);
      console.log("Attached datum", d);
    });
```

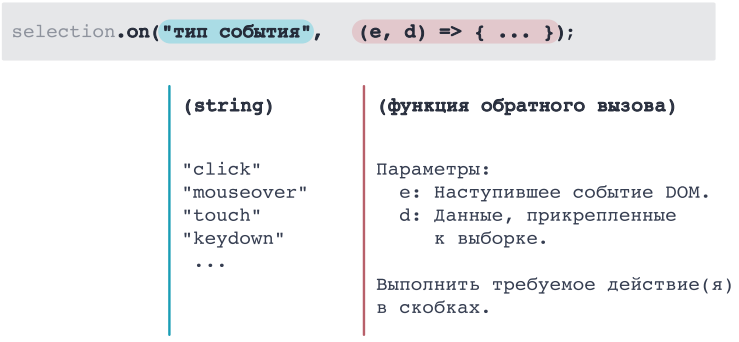


Рис. 7.2 Метод `on()` прикрепляет обработчик событий в выборку D3 и выполняет определенные действия при наступлении события

Если вывести в консоль первый параметр, полученный функцией обратного вызова (`e`), то можно увидеть, что он содержит составной объект. Такие свойства объекта, как `clientX`, `clientY`, `offsetX`, `offsetY`, `pageX` и `pageY`, представляют координаты относительно различных элементов на странице, в которых произошло событие. Тип события может быть подтвержден свойством `type`, в то время как свойство `target`, вероятно, является используемым чаще всего. Это свойство дает нам элемент, в котором произошло событие, – в данном случае `button.filter`. Оно также включает данные, прикрепленные библиотекой D3 к элементу под ключом `__data__`.

Получить доступ к данным можно более прямым способом – с помощью второго параметра, получаемого функцией обратного вызова (`d`). Если вывести в консоль второй параметр `d` и щелкнуть на кнопке с меткой «Women» (Женщины), то вы увидите объект, прикрепленный к кнопке:

```
{ id: "female", label: "Women", isActive: false }
```

Мы используем функцию обратного вызова для выполнения действий в ответ на наступившее событие. В листинге 7.2 мы сначала проверяем, что щелчок, вызвавший событие, произведен на неактивной

кнопке, так как никакие манипуляции не должны выполняться для обновления диаграммы, если это не нужно. Затем мы выполняем обход массива `filters` и обновляем свойство `isActive` каждого фильтра, исходя из нажатой кнопки. Для этого мы выбираем все кнопки с помощью D3 и используем метод `classed()`, чтобы добавить или удалить имя класса `"active"` у кнопок. В завершение мы вызываем функцию `updateHistogram()` и передаем ей идентификатор нажатой кнопки и полный набор данных. Эта функция уже объявлена в `interactions.js`, и именно здесь мы обновим гистограмму.

Листинг 7.2 Обработка активного состояния (interactions.js)

```
...
.on("click", (e, d) => {

  if (!d.isActive) {

    filters.forEach(filter => {
      filter.isActive = d.id === filter.id ?
        true : false;
    });

    d3.selectAll(".filter")
      .classed("active", filter => filter.id ===
        d.id ? true : false);

    updateHistogram(d.id, data);

  }

});
```

Проверить, что нажатая кнопка в данный момент не активна

Выполнить цикл по массиву filters и обновить свойство isActive в зависимости от нажатой кнопки

Вызвать метод classed(), чтобы добавить или удалить имя класса "active" у кнопок

Вызвать функцию filterHistogram(), которая обновит гистограмму

7.2.2 Метод classed

До сих пор в этой книге мы обрабатывали имена классов с помощью метода `attr()`, который принимает имя атрибута в первом параметре и его значение во втором. Допустим, мы хотим дать имя класса `"filter"` элементам в выборке. Для этого просто используем метод `attr()` и передаем `"class"` как имя атрибута и `"filter"` как его значение:

```
filtersSelection
  .attr("class", "filter");
```

Метод `attr()` позволяет манипулировать именами классов только как единым целым. Чтобы добавить имя класса `"active"` к выбранной кнопке, мы могли бы использовать метод `attr()`, но поскольку этот метод полностью перезаписывает атрибут, нам также придется включить начальное имя класса `"filter"` и добавить второе имя через пробел:

```
myActiveFilter
  .attr("class", "filter active");
```

К счастью, D3 предоставляет метод `classed()`, позволяющий манипулировать каждым именем класса отдельно. С его помощью можно указать, должна ли иметь выборка данное имя класса или нет. Он принимает имя класса в первом параметре и логическое значение (`true` или `false`) во втором, как показано на рис. 7.3. В нашем примере вместо повторного применения имени класса `"filter"` при каждом добавлении или удалении имени класса `"active"` мы можем использовать метод `classed()` и управлять именем класса `"active"` отдельно:

```
myActiveFilter
  .classed("active", true);
```

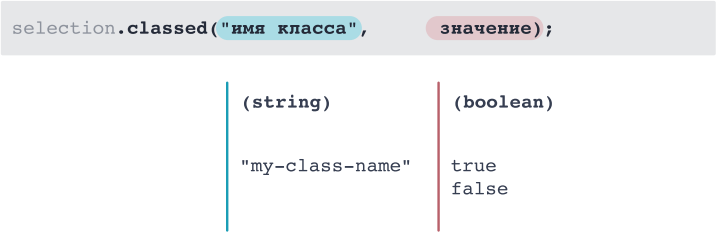


Рис. 7.3 Метод `classed()` позволяет управлять каждым именем класса в выборке по отдельности. В первом параметре он принимает имя класса, а во втором – значение `true` или `false`, указывающее, должна ли выборка иметь имя класса

7.2.3 Обновление данных в визуализации

Теперь, получив возможность перехватывать события нажатий на кнопки, мы готовы обновить гистограмму соответствующим образом. Начнем работать с функцией `updateHistogram()`, которую вызывали ранее в листинге 7.2.

Если пользователь щелкает на кнопке **Women** (Женщины), то мы должны обновить гистограмму так, чтобы она содержала только данные респондентов, идентифицирующих себя как женщины. Аналогично если пользователь нажимает кнопку **Men** (Мужчины), мы должны обновить гистограмму так, чтобы она содержала только данные респондентов, идентифицирующих себя как мужчины. Наконец, если пользователь щелкнет на кнопке **All** (Все), то гистограмма должна визуализировать весь набор данных.

В листинге 7.3 первый параметр, который получает `updateHistogram()`, – это свойство `id` данных, прикрепленных к выбранному фильтру, которое может иметь три разных значения: `"all"`, `"female"` или `"male"`. Если `id` содержит значение `"all"`, то мы сохраняем весь

набор данных (второй параметр). В противном случае фильтруем его, чтобы оставить только ответы людей выбранного пола. Новые данные сохраняются в константе `updatedData`.

Поскольку данные изменились, мы также должны пересчитать корзины, используемые для построения гистограммы. Для этого вызывается генератор корзин, созданный при конструировании исходной гистограммы, и ему передается обновленный набор данных. Полученные в ответ корзины сохраняются в константе `updatedBins`. Обратите внимание: чтобы генератор корзин был доступен из этой функции, мы добавили его в глобальные переменные, перечисленные в `shared-constants.js`.

Наконец, мы выбираем элементы прямоугольников, составляющих гистограмму, и привязываем к ним новые корзины. После этого остается только пересчитать атрибуты `y` и `height` прямоугольников с использованием шкалы `yScale`, объявленной при конструировании гистограммы.

Листинг 7.3 Обновление данных в гистограмме (interactions.js)

```
const updateHistogram = (filterId, data) => {

  const updatedData = filterId === "all"
    ? data
    : data.filter(respondent => respondent.gender
        === filterId);

  const updatedBins = binGenerator(updatedData);

  d3.selectAll("#histogram rect")
    .data(updatedBins)
      .attr("y", d => yScale(d.length))
      .attr("height", d => innerHeight -
        yScale(d.length));

};
```

Отфильтровать исходный набор данных с учетом нажатой кнопки

Связать новые корзины с элементами прямоугольников гистограммы и обновить их атрибуты `y` и `height`

Пересчитать корзины вызовом генератора корзин и передачей ему отфильтрованного набора данных

Опробовав проект теперь, вы заметите, что гистограмма обновляется после каждого щелчка на той или иной кнопке фильтра, как показано на рис. 7.4. Но смена происходит резко, без плавного перехода. Мы исправим этот недостаток в следующем подразделе.

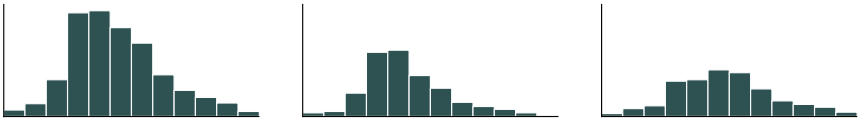


Рис. 7.4 Три состояния гистограммы: со всеми данными, только с данными по женщинам и только с данными по мужчинам

7.2.4 Создание плавных переходов

Взаимодействия обычно подразумевают изменение между двумя или более состояниями, например синий элемент становится зеленым или подсказка переходит из скрытого состояния в видимое. Обеспечение плавности переходов, т. е. анимация между состояниями А и Б, важно не только с точки зрения эстетики. Тщательно продуманные анимации и переходы могут снизить когнитивную нагрузку пользователей, помогая им наблюдать протекающие изменения, куда и что переместилось и почему. Анимации буквально помогают пользователям понять происходящее.

В D3 плавные переходы выполняются с помощью метода `transition()`, который является частью модуля d3-transition. Мы включаем метод `transition()` в цепочку непосредственно перед изменением значений атрибутов и стилей, к которым требуется применить анимацию. Например, следующий фрагмент добавляет переход в обновление данных гистограммы, реализованное в листинге 7.3. Поскольку мы добавляем вызов метода `transition()` перед обновлением атрибутов `y` и `height`, они будут анимированы оба. Переходы влияют только на свойства, изменение которых происходит после метода `transition()`:

```
d3.selectAll("#histogram rect")
  .data(updatedBins)
  .transition()
    .attr("y", d => yScale(d.length))
    .attr("height", d => innerHeight - yScale(d.length));
```

Как и CSS, библиотека D3 тоже позволяет управлять параметрами переходов. Во-первых, можно задать его длительность с помощью метода `duration()`, который принимает значение в миллисекундах. В отсутствие этого метода D3 применяет длительность по умолчанию в 250 мс.

Также можно применить задержку перед началом перехода, вызвав метод `delay()`, который тоже принимает значение в миллисекундах. Задержка по умолчанию составляет 0 мс.

Наконец, мы можем указать функцию динамики перехода (изменения скорости с течением времени). В реальной жизни объекты редко движутся с постоянной или линейной скоростью. Движение чаще начинается медленно и постепенно ускоряется, или движение начинается быстро и замедляется к концу. Эти явления можно воспроизвести с помощью функций динамики переходов. Поскольку они до определенной степени имитируют реальность, тщательно выбранные функции динамики могут сделать переходы более естественными. Вы, вероятно, уже использовали функции динамики в CSS, такие как `ease-in`, `ease-out` и `ease-in-out`.

На рис. 7.5 показана разница между линейной скоростью и наиболее распространенными функциями динамики: квадратичной, кубической, экспоненциальной, синусоидальной, круговой и полиномиальной. На каждом графике горизонтальная ось представляет время, а вертикальная – скорость изменения. Изменением может быть, например, местоположение, цвет или непрозрачность элемента. Синусоидальные и квадратичные функции динамики дают более плавное изменение скорости, а экспоненциальная – более динамичное.

Обратите внимание, что полиномиальная функция представлена на рис. 7.5 в виде области размаха. Это объясняется тем, что D3 позволяет изменять значение показателя степени, применяемого ко времени, обеспечивая целый диапазон возможностей. Значение по умолчанию – 3, что соответствует кубической функции.



Рис. 7.5 Наиболее распространенные функции динамики переходов в сравнении с линейной скоростью изменения. На каждом графике горизонтальная ось представляет время, а вертикальная – скорость изменения

В общедоступном блокноте Observable «Easing Graphs» (https://observablehq.com/@d3/easing) вы можете посмотреть, как влияют на кривую различные экспоненты. Каждая функция динамики представлена в трех форматах:

- *анимации, начинающиеся медленно и ускоряющиеся по мере приближения к концу*. Имена этих функций имеют суффикс «In», например `d3.easeQuadIn`, `d3.easeExpIn`, `d3.easeCircleIn` и т. д.;
- *анимации, начинающиеся быстро и замедляющиеся по мере приближения к концу*. Имена этих функций имеют суффикс «Out», например `d3.easeQuadOut`, `d3.easeExpOut`, `d3.easeCircleOut` и т. д.;
- *анимации, начинающиеся и заканчивающиеся медленно*. Имена этих функций имеют суффикс «InOut», например `d3.easeQuadInOut`, `d3.easeExpInOut`, `d3.easeCircleInOut` и т. д.

Если функция динамики не задана, то D3 по умолчанию использует `d3.easeCubicInOut`. В дополнение к более традиционным функциям D3 предлагает еще три функции динамики, имитирующие определенные физические реакции. Эти функции описаны ниже и представлены на рис. 7.6:

- функция динамики, воспроизводящая поведение эластичного материала, который может деформироваться и возвращаться к первоначальной форме, как резинка. Она принимает два параметра, амплитуду и период, которые могут существенно влиять на скорость изменения с течением времени. Увидеть их влияние можно по адресу https://observablehq.com/@d3/easing;
- функция динамики с кратковременным возвратом создает эффект кратковременного изменения скорости в обратном направлении. Влияние ее параметров возможно увидеть по адресу https://observablehq.com/@d3/easing;
- функция динамики с отскоком имитирует поведение подпрыгивающего мяча, прежде чем достичь полной амплитуды движения. Эта функция не имеет параметров.

Эти функции смягчения имеют по три разновидности (In, Out и InOut), упоминавшиеся выше.

> **СОВЕТ** Увидеть в действии каждую функцию динамики, предлагаемую D3, можно в блокноте «Easing Animations», разработанном Майком Бостоком (Mike Bostock) (http://mng.bz/vPg7). Если графики на рис. 7.5 и 7.6 не до конца вам понятны, то эта определенно вам поможет!



Рис. 7.6 Функции динамики, которые имитируют эластичный материал, кратковременный возврат и отскоки. На каждом графике горизонтальная ось представляет время, а вертикальная – скорость изменения

Вернемся к нашему проекту и настроим анимацию гистограммы. В следующем фрагменте мы замедляем переход, задав длительность 500 мс и применив функцию динамики `d3.easeCubicOut`. Для таких сценариев, когда пользователь щелкает на кнопке и ожидает, что что-то произойдет, предпочтительнее выбрать динамику, начинающую анимацию быстро и замедляющую ее по мере приближения к концу, поскольку она обеспечивает немедленную обратную связь для пользователя, сохраняя при этом естественность ощущений. По этой же причине мы не будем применять задержку к переходу:

```
d3.selectAll("#histogram rect")
  .data(updatedBins)
  .transition()
    .duration(500)
```

```
    .ease(d3.easeCubicOut)
    .attr("y", d => yScale(d.length))
    .attr("height", d => innerHeight - yScale(d.length));
```

Как видите, D3 позволяет легко применять плавные переходы. Главное правило, что следует запомнить (выделено на рис. 7.7): переход влияет только на свойства (атрибуты и стили), настраиваемые после метода `transition()`.

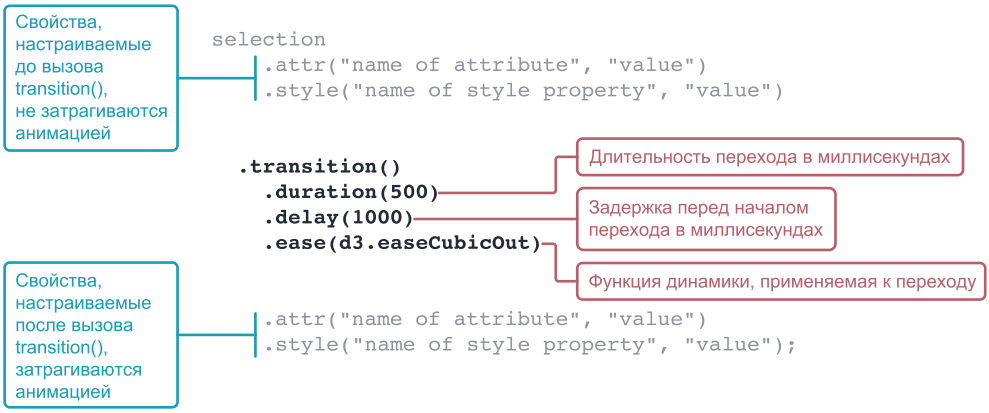


Рис. 7.7 Переходы D3 влияют только на свойства, настраиваемые после метода `transition()`. Мы можем настраивать длительность, задержку и функцию динамики переходов

7.3 Отображение дополнительной информации с помощью всплывающих подсказок

Всплывающие подсказки – едва ли не самое первое, что приходит на ум, когда мы начинаем размышлять об интерактивных визуализациях. Они позволяют добавлять дополнительную информацию, не перегружая диаграмму.

Поскольку подсказки не занимают дополнительного места, они предлагают отличный способ отображения дополнительной информации в визуализациях.

В этом разделе мы создадим два типа подсказок. Начнем с простой, более классической подсказки, которая показывает температуру на линейной диаграмме, созданной в главе 4. Затем создадим составную подсказку, которая будет следовать за указателем мыши по потоковому графику, созданному в главе 5, и сообщать информацию о продажах музыкальных записей по видам носителей за соответствующий год.

7.3.1 Создание простой подсказки

В этом разделе мы будем работать с линейной диаграммой погоды в Нью-Йорке за 2021 год, созданной в главе 4. Когда указатель мыши пересекает кружок, мы будем отображать подсказку с точной температурой, которую он представляет. Эта диаграмма показана на рис. 7.8, а как действует подсказка – по адресу http://mng.bz/46dw.

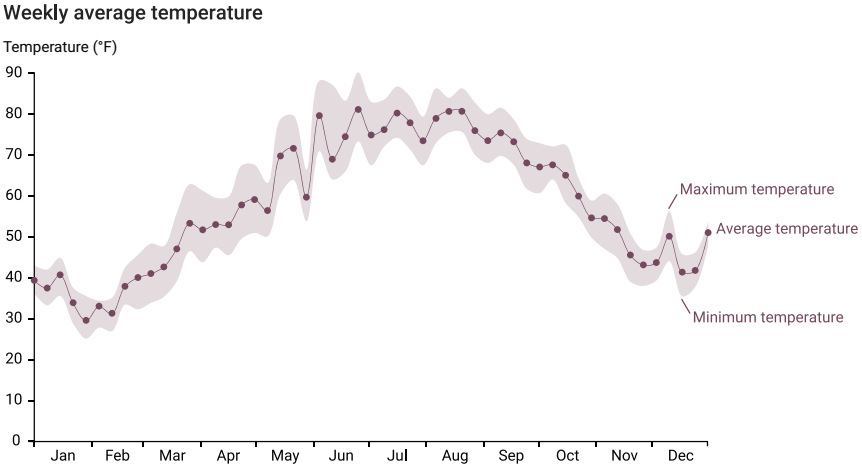


Рис. 7.8 Линейный график средней недельной температуры в Нью-Йорке за 2021 год, построенный в главе 4

> **ПРИМЕЧАНИЕ** Загрузите или клонируйте файлы из репозитория книги на GitHub (http://mng.bz/Xqjv). Папка `chapter_07/7.3.1-Simple_tooltip` содержит две подпапки с именами `start` и `end`. Откройте папку `start` в редакторе и запустите локальный веб-сервер. Если у вас возникнут трудности, обратитесь к решению в папке `end`.

В D3 создавать подсказки можно множеством способов. В этом проекте мы выполним шаги, перечисленные на рис. 7.9. Сначала мы создадим подсказку, добавив элемент группы во внутреннюю диаграмму. Эта группа будет содержать элемент прямоугольника, играющий роль фона подсказки, и текстовый элемент, отображающий текст подсказки.

Итак, начнем работу внутри функции `createTooltip()`, которую вы найдете в `interactions.js`. Обратите внимание, что эта функция уже вызывается после загрузки данных, поэтому код, который она содержит, будет выполнен автоматически.

Как показано в листинге 7.4, во внутреннюю диаграмму сначала добавляется группа с именем класса `"tooltip"`. Имя класса мы будем использовать позже для выбора подсказки и изменения ее

местоположения. Выборка с подсказкой сохраняется в константе с именем `tooltip`.

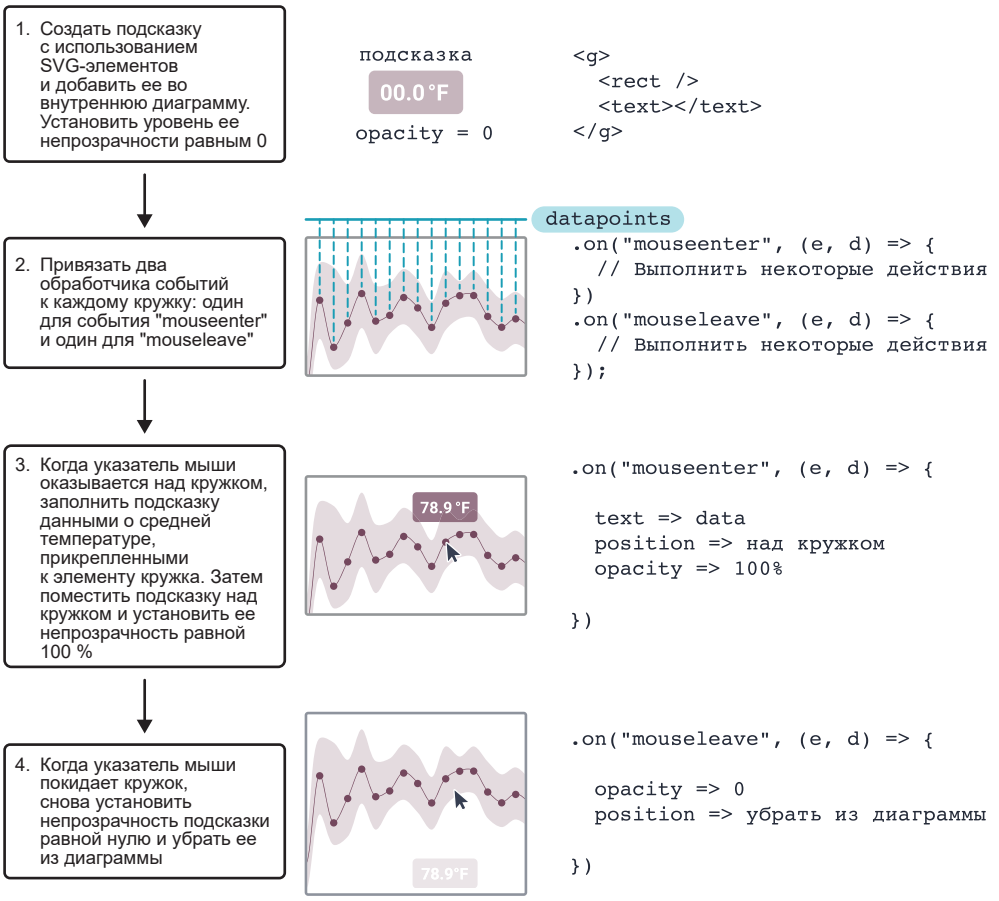


Рис. 7.9 Чтобы создать подсказку, сначала добавим элемент группы во внутреннюю диаграмму. В эту группу мы добавим прямоугольник, играющий роль фона подсказки, и текстовый элемент. Для подсказки мы установим уровень непрозрачности равным 0. Затем к кружкам диаграммы мы привяжем обработчики событий `"mouseenter"` и `"mouseleave"`. Когда указатель мыши входит в пределы кружка, мы устанавливаем текст подсказки, помещаем его над кружком и устанавливаем уровень непрозрачности равным 100 %. Когда указатель мыши покидает пределы кружка, мы устанавливаем уровень непрозрачности равным 0 и убираем подсказку из диаграммы

Затем в выборку с подсказкой добавляется элемент прямоугольника, который будет играть роль фона. Его ширина и высота задаются с помощью констант `tooltipWidth` и `tooltipHeight` со значениями 65 и 32 пикселя соответственно. Эти константы уже объявлены в `shared-constants.js`. С помощью атрибутов `rx` и `ry` углам прямоугольника придается закругление с радиусом 3 пикселя. Атрибуту

`fill` присваивается цвет `aubergine` (еще одна константа, объявленная в `shared-constants.js` и использованная ранее для рисования линейной диаграммы). Наконец, прямоугольник делается полупрозрачным путем присваивания атрибуту `fill-opacity` значения `0.75`, чтобы подсказка не загораживала диаграмму, когда она видима.

В заключение в подсказку добавляется текстовый элемент. В качестве текста устанавливается строка `"00.0°F"`. Это сделано с целью проверить, достаточно ли большой прямоугольник, чтобы вместить текст, но вообще этот шаг можно опустить. Атрибутам `text-anchor` и `alignment-baseline` присваивается значение `"middle"`, а затем текст помещается в центр прямоугольника с атрибутами `x` и `y`. Устанавливается белый цвет текста, а свойству стиля `"font-weight"` присваивается значение `900`, чтобы улучшить читаемость.

Листинг 7.4 Создание элементов всплывающей подсказки (interactions.js)

```
const createTooltip = () => {

  const tooltip = innerChart
    .append("g")
      .attr("class", "tooltip");

  tooltip
    .append("rect")
      .attr("width", tooltipWidth)
      .attr("height", tooltipHeight)
      .attr("rx", 3)
      .attr("ry", 3)
      .attr("fill", aubergine)
      .attr("fill-opacity", 0.75);

  tooltip
    .append("text")
      .text("00.0°F")
      .attr("x", tooltipWidth/2)
      .attr("y", tooltipHeight/2 + 1)
      .attr("text-anchor", "middle")
      .attr("alignment-baseline", "middle")
      .attr("fill", "white")
      .style("font-weight", 900);

}
```

Добавить группу SVG во внутреннюю диаграмму и придать ей класс с именем "tooltip". Сохранить выборку в константе с именем tooltip

Добавить элемент прямоугольника в выборку. Настроить его ширину, высоту и цвет заливки с помощью констант, уже объявленных в shared-constants.js. Закруглить углы с помощью атрибутов `rx` и `ry` и сделать прямоугольник полупрозрачным, присвоив атрибуту fill-opacity значение 0.75

Добавить текстовый элемент. Присвоить ему текст "00.0°F" (необязательно) и отцентрировать в прямоугольнике. Задать белый цвет и установить свойство стиля font-weight равным 900, чтобы улучшить читаемость

Сохраните проект, откройте его в браузере, и вы увидите подсказку в верхнем левом углу диаграммы, как показано на рис. 7.10.

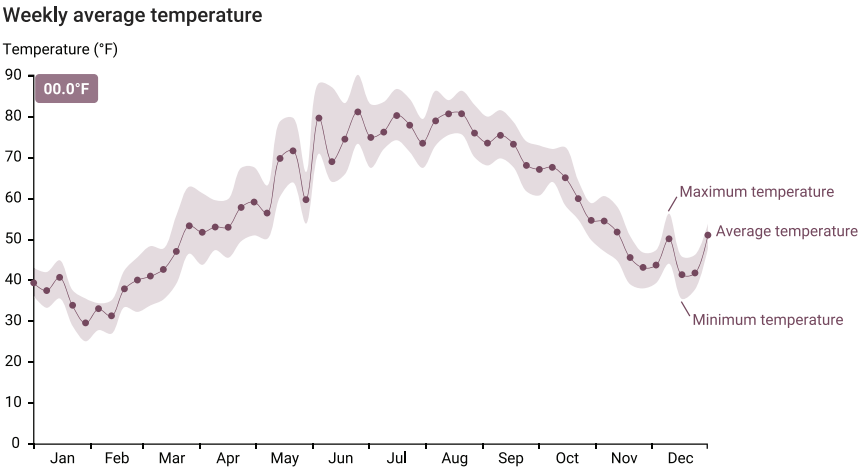


Рис. 7.10 Подсказка, добавленная во внутреннюю диаграмму

Теперь установим непрозрачность подсказки в значение 0, чтобы скрыть ее от пользователя. Через несколько мгновений мы заставим ее появляться, когда указатель мыши будет оказываться над кружками на диаграмме:

```
const tooltip = innerChart
  .append("g")
    .attr("class", "tooltip")
    .style("opacity", 0);
```

Второй шаг рабочего процесса, показанного на рис. 7.9, – подключение двух обработчиков событий к кружкам линейной диаграммы. Мы подключаем их в функции `handleMouseEvents()`, как показано в листинге 7.5, которая объявлена в `interactions.js`.

Для этого сначала выбираются все элементы кружков во внутренней диаграмме и добавляются два вызова метода `on()`. Первым вызовом `on()` подключается обработчик события `mouseenter`, возникающего, когда указатель мыши входит в область, занимаемую кружком, а вторым – обработчик события `mouseleave`, возникающего, когда указатель мыши покидает область, занимаемую кружком. Функция обратного вызова, что передается методу `on()` во втором параметре (собственно обработчик события), принимает два параметра: `e` – событие DOM и `d` – данные, привязанные к кружку, где возникло событие. Попробуйте вывести их в консоли и исследуйте их содержимое. Вы заметите, что в данных передается соответствующая запись из исходного набора.

Листинг 7.5 Добавление обработчиков событий (interactions.js)

```
const handleMouseEvents = () => {

  innerChart.selectAll("circle")
    .on("mouseenter", (e, d) => {

      console.log("DOM event", e);
      console.log("Attached datum", d);

    })
    .on("mouseleave", (e, d) => {

    });

}
```

Выбрать все кружки во внутренней диаграмме

Привязать к кружкам обработчик событий входа указателя мыши

Привязать к кружкам обработчик событий выхода указателя мыши

Теперь, зная, когда указатель мыши входит в область, занимаемую кружком, мы можем сделать подсказку видимой, выполнив следующие действия:

1. Записать в текст подсказки среднюю температуру из данных, прикрепленных к элементу кружка.
2. Поместить подсказку над кружком с помощью события DOM.
3. Установить непрозрачность подсказки равной 100 %.

Код в листинге 7.6 выбирает текстовый элемент внутри подсказки и записывает в него текст со средней температурой, указанной в полученных данных. Мы использовали метод `d3.format()`, чтобы ограничить количество цифр тремя.

Затем определяется местоположение кружка под указателем с помощью его атрибутов `cx` и `cy`. Свойство `target` события DOM возвращает атрибуты `circle` и `cy`. Эти значения используются для помещения подсказки над выбранным кружком и центрирования по горизонтали. Наконец, к свойству непрозрачности подсказки применяется метод `transition()` и устанавливается продолжительность перехода равной 200 мс.

Листинг 7.6 Создание всплывающей подсказки (interactions.js)

```
innerChart.selectAll("circle")
  .on("mouseenter", (e, d) => {

    d3.select(".tooltip text")
      .text(`${d3.format(".3")(d.avg_temp_F)}°F`);
```

Выбрать текстовый элемент подсказки и изменить его текст, чтобы он отражал температуру, указанную в полученных данных. Использовать метод d3.format(), чтобы ограничить количество цифр

```
    const cx = e.target.getAttribute("cx");
    const cy = e.target.getAttribute("cy");
    d3.select(".tooltip")
      .attr("transform", `translate(${cx -
        0.5*tooltipWidth},
        ➥ ${cy - 1.5*tooltipHeight})`)
      .transition()
        .duration(200)
        .style("opacity", 1);
});
```

Найти местоположение кружка под указателем с помощью его атрибутов `cx` и `cy`. Поместить подсказку над кружком с помощью атрибута `transform`

Установить непрозрачность подсказки равной 100 %, задав длительность перехода равной 200 мс

Сохраните проект и обратите внимание, что теперь подсказка отображается над кружками, над которыми оказывается указатель мыши, как показано на рис. 7.11.

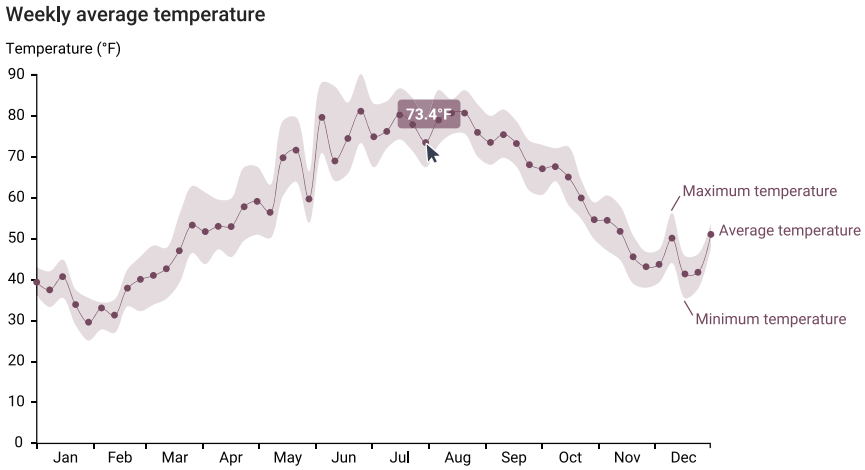


Рис. 7.11 Когда мышь входит в область, занимаемую кружком, над ним появляется подсказка с соответствующей температурой

> **СОВЕТ** При создании подсказок старайтесь не загораживать соседние маркеры, которыми в данном случае являются соседние кружки. Для этого можно, например, сделать фон подсказки полупрозрачным, как мы сделали в этом проекте.

В качестве последнего шага реализуем исчезновение подсказки, когда указатель мыши покидает область, занимаемую кружком. В функции обработчика, вызываемого по событию `"mouseleave"`, мы выбираем подсказку и меняем ее непрозрачность обратно на 0. Затем убираем подсказку из диаграммы. Подсказка, находящаяся над кружком, не позволит обнаружить события мыши, даже если она не

видна. В листинге 7.7 мы просто перемещаем ее вниз на 500 пикселей, но при желании вы можете использовать любое другое направление и расстояние перемещения.

Листинг 7.7 Сокрытие подсказки (interactions.js)

```
...
.on("mouseleave", (e, d) => {

  d3.select(".tooltip")
    .style("opacity", 0)
    .attr("transform", `translate(0, 500)`);

});
```

Вот и все – теперь наша диаграмма содержит полноценную всплывающую подсказку!

7.3.2 Разработка составной подсказки

Потоковые графики содержат области, наложенные на переменную базовую линию, что затрудняет чтение точных значений. Однако эту дополнительную информацию можно предоставить с помощью подсказки. В этом разделе мы создадим составную подсказку, которая будет следовать за указателем мыши при его перемещении над потоковым графиком, созданным в главе 5, и показывать объемы продаж записей музыки на каждом типе носителей за соответствующий год. Эту подсказку можно увидеть на рис. 7.12 и опробовать онлайн по адресу http://mng.bz/QRGG.

> **СОВЕТ** Возможно, вы заметите, что порядок расположения областей в потоковом графике отличается от порядка в главе 5. Мы изменили его, чтобы он соответствовал порядку типов носителей в подсказке. Такие незначительные улучшения могут помочь пользователям сопоставлять информацию, которую они видят на потоковом графике, с метками подсказки.

Как и в предыдущем случае, создать подсказку можно множеством способов. В этом проекте мы будем следовать шагам, перечисленным на рис. 7.13.

> **ПРИМЕЧАНИЕ** Загрузите или клонируйте файлы из репозитория книги на GitHub (http://mng.bz/Xqjv). Папка `chapter_07/7.3.2-Compound_tooltip` содержит две подпапки с именами `start` и `end`. Откройте папку `start` в редакторе и запустите локальный веб-сервер. Если у вас возникнут трудности, обратитесь к решению в папке `end`.

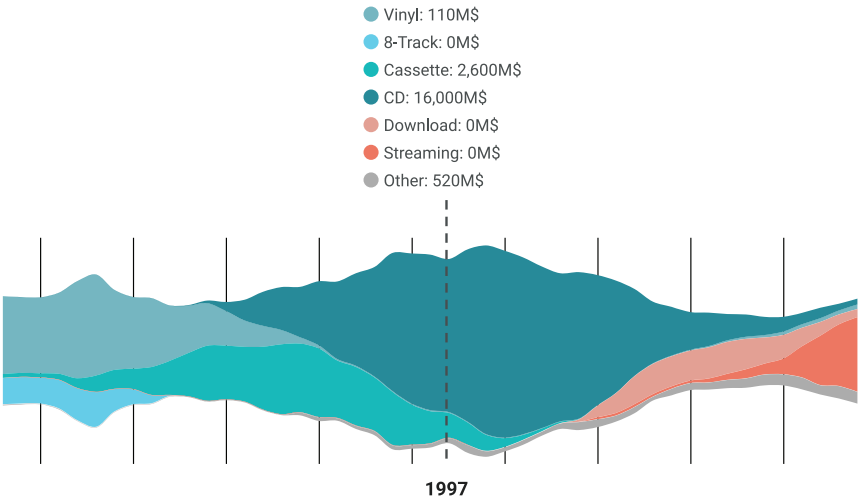


Рис. 7.12 Потоковый график, визуализирующий 40 лет продаж музыкальных записей по типам носителя. Подсказка следует за указателем мыши, показывая год и объемы продаж

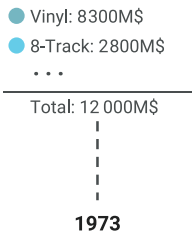
Первый шаг на рис. 7.13 заключается в создании элементов, составляющих подсказку. Создаваться элементы будут в функции `createTooltip()`, которая уже объявлена в `interactions.js` и вызывается из `load-data.js`. Она получает набор данных и использует ряд вспомогательных констант, определяющих значения для свойств `width`, `height`, `color` и `line-height` подсказки.

Код в листинге 7.8 сначала добавляет во внутреннюю диаграмму элемент группы с именем класса `"tooltip"`. Имя класса мы будем использовать позже для выбора подсказки и изменения ее местоположения.

Затем в выборку с подсказкой добавляется вертикальная линия. Горизонтальная координата линии устанавливается равной 0, а сама линия простирается по вертикали от нижней части внутренней диаграммы до 30 пикселей над ней. Толщина линии задается равной 2 пикселям, и с помощью атрибута `stroke-dasharray` со значением `"6 4"` устанавливается стиль отображения линии в виде пунктира. Черточки будут иметь длину 6 пикселей и разделяться пустым пространством протяженностью в 4 пикселя.

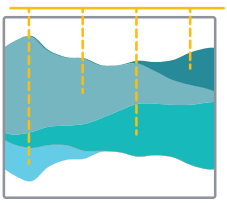
Далее добавляется текстовый элемент и размещается под линией. Здесь будет отображаться год при горизонтальном перемещении подсказки. На данный момент подсказка находится на левом конце потокового графика, поэтому в ней отображается 1973 – первый год в наборе данных, полученный с помощью `d3.min()`. Мы присваиваем этому тексту класс с именем `"tooltip-year"`, чтобы иметь возможность выбирать и обновлять его по мере перемещения подсказки.

1. Создать подсказку с элементами SVG и добавить ее во внутреннюю диаграмму. Поместить подсказку над первым годом и установить соответствующие значения с объемами продаж




```
<g>
  <line />
  <text>1973</text>
  <g>
    <text>
      <tspan>Vinyl: $</tspan>
      <tspan>8-Track: $</tspan>
      ...
    </text>
    <circle />
    <circle />
    ...
  </g>
</g>
```

2. Привязать обработчик событий "mousemove" к элементам path потокового графика

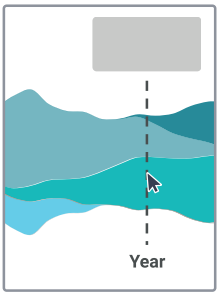



```
.on("mousemove", e => {

  // Выполнить некоторые действия

});
```

3. Когда указатель мыши перемещается вдоль потокового графика, получить его горизонтальную координату (относительно потокового графика) с помощью метода `d3.pointer()`. Переместить подсказку в место, где находится указатель мыши

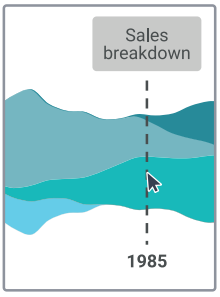



```
.on("mousemove", e => {

  x => d3.pointer(e)[0]

})
```

4. Передать горизонтальную координату указателя мыши методу invert() шкалы xScale, чтобы получить год. Найти в исходном наборе данных информацию, соответствующую году, и заполнить метки в подсказке




```
.on("mousemove", e => {

  ...
  Year => xScale.invert(x)
  Sales => данные за год

})
```

Рис. 7.13 Шаги по созданию составной подсказки и организации ее взаимодействия с мышью

Листинг 7.8 Добавление вертикальной линии и года (interactions.js)

```
const createTooltip = (data) => {

  ...

  const tooltip = innerChart
    .append("g")
      .attr("class", "tooltip");

  tooltip
    .append("line")
      .attr("x1", 0)
      .attr("x2", 0)
      .attr("y1", -30)
      .attr("y2", innerHeight)
      .attr("stroke", textColor)
      .attr("stroke-width", 2)
      .attr("stroke-dasharray", "6 4");

  const firstYear = d3.min(data, d => d.year);
  const tooltipYear = tooltip
    .append("text")
      .attr("class", "tooltip-year")
      .attr("x", 0)
      .attr("y", innerHeight + 25)
      .style("font-size", "16px")
      .style("font-weight", 700)
      .style("fill", textColor)
      .attr("text-anchor", "middle")
      .text(firstYear);

};
```

Добавить группу SVG во внутреннюю диаграмму и присвоить ей имя класса "tooltip"

Добавить пунктирную вертикальную линию. Расположить ее на левом краю потокового графика

Добавить текстовый элемент и поместить его под вертикальной линией. Здесь будет выводиться текущий год по мере перемещения подсказки. На данный момент в ней указан первый год в наборе данных, 1973

В листинге 7.9 формируется текст подсказки с объемами продаж разных типов носителей и отображается над вертикальной линией. Сначала мы добавляем группу SVG в подсказку и перемещаем ее влево на половину ширины подсказки, а затем вверх. Эта группа сохраняется в константе с именем `tooltipContent`. Затем добавляем текстовый элемент в группу, даем ему имя класса `"tooltip-content"` и сохраняем в константе `tooltipText`. Именно в этом элементе будут отображаться объемы продаж.

После извлечения объекта с объемами продаж за первый год (1973) мы обходим в цикле массив `formatsInfo`, объявленный в `shared-constants.js`. Для каждого типа носителя мы добавляем в `tooltipText` элемент `tspan`. Копируем в него текст из свойства `label` типа носителя и объем продаж в миллионах долларов, отформатированный с разделением групп разрядов и не менее одной значащей цифры (`d3.format(",.1r")`). Каждый элемент `span` располагается под предыдущим с установкой его атрибута `y`.

Наконец, все там же внутри цикла, слева от каждого `tspan` мы добавляем кружок с радиусом 6 пикселей и цветом заливки, соответствующим типу носителя. Эти кружки будут играть роль легенды и сделают содержимое подсказки более понятным.

Листинг 7.9 Добавление информации с объемами продаж (interactions.js)

```
const createTooltip = (data) => {

  ...

  const tooltipContent = tooltip
    .append("g")
      .attr("transform", `translate(${-1 *
        tooltipWidth/2},
➥   ${-1 * margin.top + 30})`);

  const tooltipText = tooltipContent
    .append("text")
      .attr("class", "tooltip-content")
      .style("font-size", "14px")
      .style("font-weight", 500)
      .style("fill", textColor);

  const dataFirstYear = data.find(item => item.year === firstYear);

  formatsInfo.forEach((format, i) => {

    tooltipText
      .append("tspan")
        .attr("class", `sales-${format.id}`)
        .attr("x", 0)
        .attr("y", i * textLineHeight)
        .text(`${format.label}:
➥         ${d3.format(",.1r")(dataFirstYear[
            format.id])}M$`);

    tooltipContent
      .append("circle")
        .attr("cx", -10)
        .attr("cy", i * textLineHeight - 5)
        .attr("r", 6)
        .attr("fill", format.color);

  });

};
```

Добавьте группу SVG в подсказку. Она будет содержать текст с объемами продаж и кружки

Добавить текстовый элемент

Цикл по типам носителей. Для каждого типа добавить элемент tspan и присвоить ему текст с названием типа носителя и объемом продаж за первый год. Добавить также кружок, задав цвет заливки, соответствующий текущему типу носителя

По завершении подсказка должна выглядеть, как показано на рис. 7.14. Обратите внимание, что в 1973 году на рынке доминировали виниловые пластинки, а за ними следовали 8-дорожечные магнитофонные пленки.

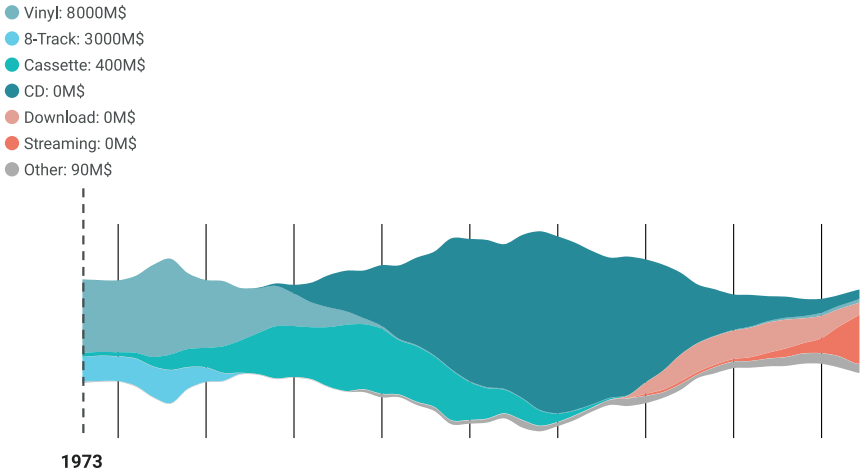


Рис. 7.14 Подсказка, размещенная в течение первого года

Создав подсказку, можно сделать следующий шаг: подключить обработчики событий к элементам `path` потокового графика и начать перемещать подсказку вместе с мышью! Для этого отыщите функцию `handleMouseEvents()`, объявленную в `interactions.js`. В листинге 7.10 мы сначала выбираем все элементы `path` в потоковом графике и используем метод `on()` для подключения обработчика событий. Нас интересует событие DOM `mousemove`, которое будет возникать при перемещении мыши вдоль потокового графика.

Затем нужно найти позицию указателя мыши и переместить подсказку туда. В D3 для этого имеется удобный метод `d3.pointer()`, который принимает событие DOM в качестве аргумента и возвращает его координаты относительно цели – в данном случае относительно элемента `path`. Этот метод является частью модуля d3-selection и возвращает горизонтальные и вертикальные координаты выборки в массиве. Поскольку нас интересует только горизонтальная координата, мы получаем ее с помощью выражения `d3.pointer(e)[0]`, а затем перемещаем подсказку.

Листинг 7.10 Перемещение подсказки (interactions.js)

```
const handleMouseEvents = (data) => {

  d3.selectAll(".areas-container path")
    .on("mousemove", e => {
```

Подключить обработчик событий "mousemove" к элементам path потокового графика

```
      const xPosition = d3.pointer(e)[0];
      d3.select(".tooltip")
        .attr("transform", `translate(
          ${xPosition}, 0)`);

    });

};
```

Получить горизонтальную координату указателя мыши с помощью d3.pointer() и переместить подсказку

Если навести указатель мыши на потоковый график, то подсказка должна последовать вслед за ним. Круто! Давайте теперь организуем обновление текста подсказки по мере перемещения указателя мыши и начнем с года. Чтобы узнать год, соответствующий местоположению мыши, можно использовать шкалу `xScale`, отвечающую за позиционирование элементов на горизонтальной оси. До сих пор в этой книге мы всегда передавали шкале значения из предметной области, чтобы получить соответствующее значение из диапазона. Однако непрерывные шкалы поддерживают обратное преобразование! Обратите внимание, что в листинге 7.11 вызывается метод `xScale.invert(xPosition)`. Методу `invert()` можно передать значение из диапазона (горизонтальную координату в пределах потокового графика) и получить соответствующее значение из предметной области – год. Поскольку `xScale` – это непрерывная шкала, она будет возвращать числа с плавающей точкой. Так как нам нужно, чтобы год был целым числом, мы передаем значение, возвращаемое вызовом `xScale.invert(xPosition)`, методу `Math.round()`. Затем мы выбираем текстовый элемент, отображающий год (с классом `"tooltip-year"`), и обновляем его, записывая текст с номером года.

Чтобы обновить текст с объемами продаж по типам носителей, мы сначала отыскиваем данные, связанные с позицией мыши. Затем выполняем цикл по типам носителей, для каждого выбираем соответствующий элемент `tspan` и обновляем объем продаж в соответствии с данными за год.

Листинг 7.11 Обновление текста подсказки (interactions.js)

```
const handleMouseEvents = (data) => {

  d3.selectAll(".areas-container path")
    .on("mousemove", e => {

      ...

      const year = Math.round(xScale.invert(xPosition));
      d3.select(".tooltip-year").text(year);
```

С помощью xScale.invert() найти год, соответствующий горизонтальному положению указателя мыши, и записать его в подсказку

```
      const yearData = data.find(item =>
        item.year === year);
      formatsInfo.forEach(format => {
        d3.select(`.sales-${format.id}`)
          .text(`${format.label}:
➥           ${d3.format(",.1r")(yearData[format.id])}M$`);
      });

    });

};
```

Найти данные, соответствующие этому году, и обновить объемы продаж

СОВЕТ Если взаимодействие нестандартное, дайте пользователям небольшую подсказку. Мы, к примеру, добавили текст инструкции над потоковым графиком.

7.4 Анимация появления, обновления и исчезновения элементов визуализации

Выше в этой главе мы обновляли данные в гистограмме щелчком на кнопке. В том примере количество элементов SVG в визуализации оставалось неизменным: 12 прямоугольников, высота которых менялась в зависимости от количества точек данных, которые они представляли. Но что, если обновление данных подразумевает появление или исчезновение элементов SVG? В этом разделе мы рассмотрим данный вопрос, построив диаграмму рассеяния.

Диаграмма рассеяния, которую мы создадим, отображает набор данных с видами китообразных и показана на рис. 7.15 (онлайн-версия доступна по адресу http://mng.bz/Xq8v). Каждый круг на диаграмме представляет вид китообразных. Горизонтальное местоположение кругов соответствует предполагаемой популяции вида, а вертикальное – среднему размеру животного в метрах. Площадь кругов пропорциональна весу кита, а цвет обозначает природоохранный статус, присвоенный Международным союзом охраны природы: от «вызывающий наименьшие опасения» (зеленый) до «находящийся в критическом состоянии» (красный).

Над диаграммой рассеяния отображаются четыре кнопки, позволяющие фильтровать диаграмму. По умолчанию в визуализации отображаются все виды и выбрана кнопка **All Cetaceans** (Все китообразные). Мы можем фильтровать китообразных по региону, где они обитают: в северном полушарии, в южном полушарии или мигрирующие через оба полушария. Каждый раз, когда выбирается фильтр, круги, исчезающие из диаграммы рассеяния, соскальзывают вниз, а круги, появляющиеся на диаграмме, выпадают сверху. Чуть ниже мы покажем, как создать этот эффект.

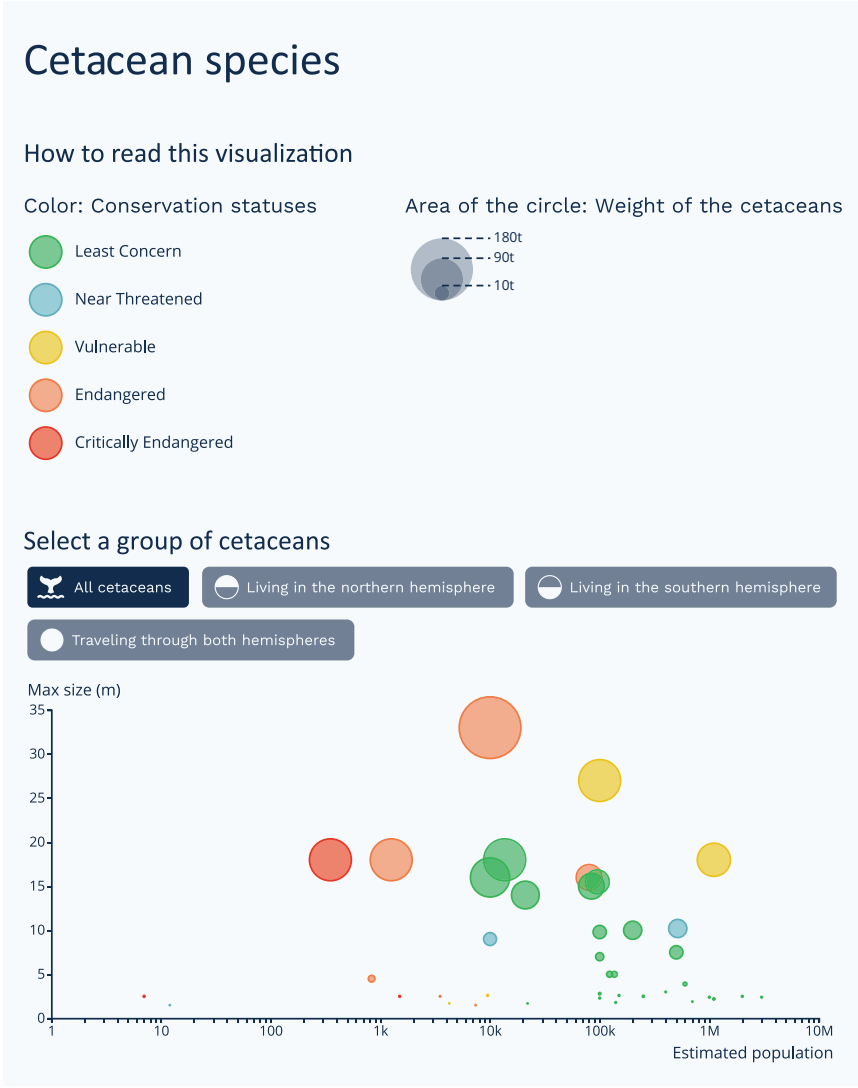


Рис. 7.15 Визуализация видов китообразных, которую мы построим в этом разделе

ПРИМЕЧАНИЕ Загрузите или клонируйте файлы из репозитория книги на GitHub (http://mng.bz/Xqjv). Папка `chapter_07/7.4-Refining_data_updates` содержит две подпапки с именами `start` и `end`. Откройте папку `start` в редакторе и запустите локальный веб-сервер. Если у вас возникнут трудности, обратитесь к решению в папке `end`.

7.4.1 Построение диаграммы рассеяния

Если вы откроете папку `start` проекта этого раздела в редакторе и запустите локальный веб-сервер, то увидите, что кнопки фильтров уже на месте и нам остается только построить диаграмму рассеяния. Для этого откройте файл `scatterplot.js` и найдите функцию `drawScatterplot()`. Эта функция уже вызывается из `load-data.js` и получает загруженные данные. Загляните в файл `README` внутри `chapter_07/7.4-Refining_data_updates`, там описывается формат набора данных и различных файлов этого проекта.

В `scatterplot.js` мы уже добавили контейнер SVG и внутреннюю диаграмму, следуя стратегии, представленной в главе 4. Теперь объявим шкалы. В этом проекте нам понадобятся четыре шкалы, которые мы уже объявили в `shared-constants.js`, чтобы сделать их глобально доступными:

- `yScale` – линейная шкала, отвечающая за распределение размеров китообразных по оси `y`;
- `colorScale` – порядковая шкала, определяющая цвет кругов в зависимости от природоохранного статуса представляемых ими видов;
- `xScale` – логарифмическая шкала распределения популяции вида по оси `x`;
- `rScale` – радиальная шкала для расчета площади кругов на основе веса представителей вида.

Нам нужно определить предметные области и диапазоны шкал на основе данных, что мы сделаем в `scatterplot.js`. Начнем с линейной шкалы, поскольку с ней мы уже знакомы. Предметная область `yScale` является непрерывной и простирается от 0 до максимального размера китообразных в наборе данных, который мы отыскиваем с помощью `d3.max()`. Шкала преобразует предметную область в непрерывный диапазон, простирающийся от `innerHeight` внизу до 0 вверху внутренней диаграммы (вы должны помнить, что в мире SVG вертикальная ось направлена сверху вниз!). Наконец, в цепочку создания шкалы мы добавляем метод `nice()`, чтобы гарантировать, что ось `y` заканчивается круглым значением:

```
const maxSize = d3.max(data, d => d.max_size_m);
yScale = d3.scaleLinear()
  .domain([0, maxSize])
  .range([innerHeight, 0])
  .nice();
```

Затем займемся цветовой шкалой, потому что порядковые шкалы мы уже обсуждали в главе 5. Порядковая шкала отображает дискретную предметную область в дискретный диапазон. Здесь предметная область – это массив идентификаторов природоохранного статуса, созданный с помощью JavaScript-метода `map()` и массива `conservationStatuses`, уже объявленного в `shared-constants.js`. Для генерации массива цветов мы используем тот же метод `range`:

```
colorScale = d3.scaleOrdinal()
  .domain(conservationStatuses.map(s => s.id))
  .range(conservationStatuses.map(s => s.color));
```

Для создания двух других шкал потребуются дополнительные объяснения, поскольку с ними мы встречаемся впервые. Мы сделаем это в следующих подразделах.

Логарифмическая шкала

В нашем проекте самый малочисленный вид китообразных – дельфины байджи, численность которых в настоящее время оценивается от 0 до 13 особей (последняя экспедиция по оценке популяции не обнаружила ни одной), а самый многочисленный – пантропические пятнистые дельфины, численность которых оценивается в 3 млн особей. Это огромная разница!

Если бы мы представляли эти популяции в линейном масштабе, то виды с наименьшей численностью были бы сжаты в точку и их фактические значения было бы невозможно прочитать. Но взгляните, насколько легко читаются численности на логарифмической оси на рис. 7.16, где значение в каждой метке увеличивается в 10 раз.

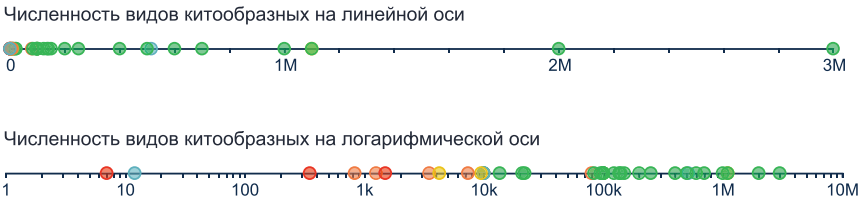


Рис. 7.16 На линейной оси виды с небольшой численностью получаются скученными, и их значения почти невозможно прочитать. Логарифмическая шкала помогает решить эту проблему, представляя популяции с десятикратным увеличением (1, 10, 100, 1k, 10k, 100k, 1M и 10M)

В D3 логарифмические оси создаются с помощью логарифмической шкалы `d3.scaleLog()`, которая возвращает логарифмическое представление непрерывной предметной области. Эта шкала особенно удобна, когда исходные данные имеют значительный разброс по величине.

> **СОВЕТ** Улучшение читаемости, которое дает логарифмическая шкала, достигается ценой потери ощущения, насколько малы некоторые значения по сравнению с большими. Логарифмические шкалы занимают важное место в практике, поэтому следует знать об этом их недостатке.

Следующий фрагмент кода задает предметную область и выходной диапазон логарифмической шкалы. Предметная область непрерывна и принимает массив с минимальным и максимальным значениями численности. Обратите внимание, что мы задали минимальное значение 1, а не 0, потому что, как вы наверняка помните из уроков математики, логарифм нуля не определен (точнее, сдвигается в сторону $-\infty$). Диапазон охватывает ширину внутренней диаграммы. Мы также добавляем в цепочку метод `nice()`, чтобы ось заканчивалась красивым круглым значением:

```
const maxPopulation = d3.max(data, d => d.global_population_estimate);
xScale = d3.scaleLog()
  .domain([1, maxPopulation])
  .range([0, innerWidth])
  .nice();
```

Радиальная шкала

До сих пор в этой книге мы выводили значения данных только на одномерные графические метки, такие как длина столбца в столбчатой диаграмме или горизонтальное положение на временной шкале. Но при визуализации данных для представления величин нередко приходится использовать площадь круга. Мы не можем использовать линейную шкалу для отображения одномерных данных – в нашем случае веса китообразных – на двумерной графической метке (каковой является площадь круга).

Давайте для начала вспомним уравнение площади круга (A), которая равна произведению «пи» ($\pi = 3{,}1416\ldots$) на квадрат радиуса. Радиус умножается сам на себя; его влияние на площадь круга квадратично, а не линейно:

$$A = \pi r^2.$$

Чтобы отобразить линейное значение на графическом значке, определяемом его радиусом, например на круге, можно воспользоваться радиальной шкалой. Как показано на рис. 7.17, круги, размер которых соответствует линейной шкале, создают визуальное искажение, преувеличивающее разницу в площади кругов, представляющих 10, 100 и 200 тонн. С другой стороны, радиальная шкала внутренне возводит диапазон в квадрат и лучше подходит для измерения окружностей.

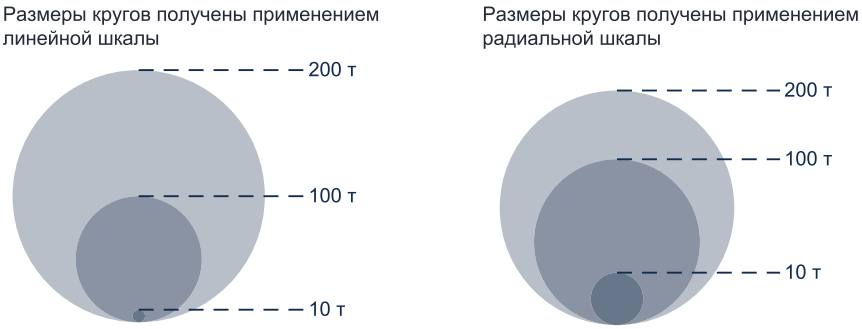


Рис. 7.17 Вес китообразных представлен кругами. Слева круги нарисованы с применением линейной шкалы, а справа – радиальной. Обратите внимание, что линейная шкала создает визуальное искажение, преувеличивая разницу между площадями кругов, соответствующих разным весам

Следующий фрагмент кода объявляет радиальную шкалу. И предметная область, и диапазон являются непрерывными – предметная область простирается от 0 до максимального веса в наборе данных, а диапазон – от 0 до 45, радиуса наибольшего круга, который будет присутствовать в визуализации:

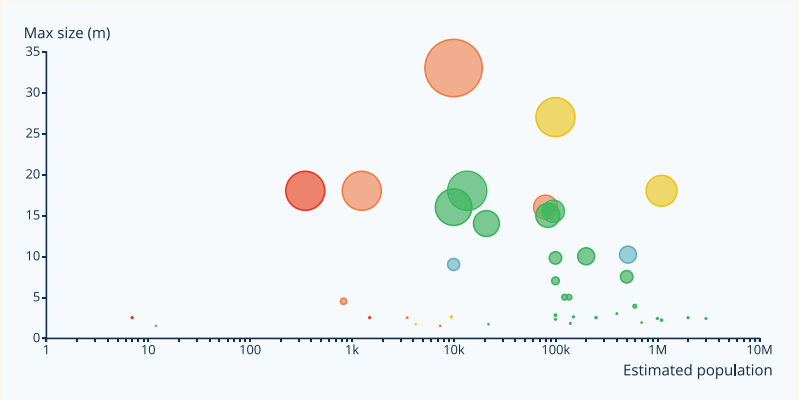
```
const maxWeigth = d3.max(data, d => d.max_weight_t);
rScale = d3.scaleRadial()
  .domain([0, maxWeigth])
  .range([0, 45]);
```

Упражнение: создайте ось и добавьте круги в диаграмму рассеяния

После объявления шкал можно приступать к созданию диаграммы рассеяния. У вас уже есть все необходимые для этого знания. Если вам нужны дополнительные наводящие подсказки, то выполните следующие шаги:

1. Объявите генератор для нижней оси (`d3.axisBottom()`) и передайте ему `xScale`. Добавьте ось во внутреннюю диаграмму.
2. Объявите генератор для левой оси (`d3.axisLeft()`) и передайте ему `yScale`. Добавьте ось во внутреннюю диаграмму.
3. Добавьте два текстовых элемента в контейнер SVG, которые будут играть роль меток осей: «Estimated population» (Оценка численности) и «Max size (m)» (Максимальный размер (м)).
4. Используя шаблон привязки данных, добавьте во внутреннюю диаграмму по одному кругу для каждого китообразного.
5. Задайте атрибуты `cx`, `cy`, `r` и `fill` кругов, используя логарифмическую, линейную, радиальную и цветовую шкалы, объявленные ранее.
6. При желании можете установить непрозрачность заливки равной 60 % и добавить к кругам обводку толщиной в 2 пикселя.

По завершении диаграмма рассеяния должна выглядеть, как показано на следующем рисунке.



Законченная диаграмма рассеяния

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в листинге D.5 и в папке `7.4-Refining_data_updates/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

> **СОВЕТ** Для определения размеров кругов можно также использовать шкалу `d3.scaleSqrt()`, которая является степенной шкалой, применяющей показатель 0,5. Результаты получились бы такими же.

Чтобы сгенерировать легенду проекта, раскомментируйте функцию `populateLegend()` в `load-data.js`. Мы рекомендуем вам исследовать код в `legend.js`, поскольку вы как разработчик визуализаций на D3 обязательно столкнетесь с необходимостью создания легенд. Мы не будем здесь подробно объяснять реализацию легенды, так как она использует концепции, с которыми вы уже знакомы.

7.4.2 Фильтрация диаграммы рассеяния

Теперь мы добавим фильтрацию диаграммы рассеяния. Фильтрация будет выполняться после щелчка на одной из кнопок над диаграммой. Мы также поговорим о том, как управлять анимацией, когда круги появляются в диаграмме или покидают ее.

Сначала откройте файл `interactions.js`. Кнопки фильтров уже созданы в функции `populateFilters()`. Мы будем обрабатывать щелчки на кнопках внутри функции `handleClickOnFilter()`. В листинге 7.12

мы сначала выбираем все кнопки с классом `"filter"` и привязываем к ним обработчик событий. Этот обработчик ждет события `"click"` и получает данные, прикрепленные к выбранному фильтру. В роли данных выступает соответствующий объект из массива `cetaceanFilters`, который находится в `shared-constants.js`.

Если нажатая кнопка еще не была выбрана, то мы обновляем свойства `isActive` в массиве `cetaceanFilters` и обновляем имя класса `"active"` кнопок. В зависимости от нажатой кнопки мы фильтруем исходный набор данных, чтобы на диаграмме остались только китообразные, соответствующие выбору.

Наконец, мы используем шаблон привязки данных для обновления количества кругов на экране, перемещения кругов в их позиции, вычисления их радиусов и выбора цвета – все эти трансформации происходят с плавным переходом. Как видите, стратегия пока очень похожа на ту, что использовалась в разделе 7.2.

Листинг 7.12 Обновление диаграммы рассеяния (interactions.js)

```
const handleClickOnFilter = (data) => {

  d3.selectAll(".filter")
    .on("click", (e, datum) => {

      if (!datum.isActive) {
        cetaceanFilters.forEach(h => h.isActive =
          h.id === datum.id
            ? true : false);

        d3.selectAll(".filter")
          .classed("active", d => d.id ===
            datum.id ? true : false);

      const updatedData = datum.id === "all"
        ? data
        : data.filter(d => d.hemisphere === datum.id);

      innerChart
        .selectAll("circle")
        .data(updatedData)
        .join("circle")
        .transition()
          .attr("class", "cetacean")
          .attr("cx", d => xScale(
            d.global_population_estimate))
          .attr("cy", d => yScale(d.max_size_m))
          .attr("r", d => rScale(d.max_weight_t))
          .attr("fill", d => colorScale(d.status))
          .attr('fill-opacity', 0.6)
          .attr("stroke", d => colorScale(d.status))
          .attr("stroke-width", 2);
```

Выбрать все кнопки с классом "filter" и привязать к ним обработчик события "click"

Убедиться, что нажатая кнопка еще не выбрана

Обновить свойства isActive в массиве cetaceanFilters и имя класса "active" кнопок

Отфильтровать исходный набор данных, чтобы сохранить только китообразных, соответствующих выбору

Использовать шаблон привязки данных для обновления количества кругов на экране. Задать их положение, радиус и цвет

```
      }
    });
};
```

Сохраните проект, откройте его в браузере и понажимайте на кнопки фильтров. Вы увидите, что количество кругов в диаграмме рассеяния меняется в зависимости от выбранного фильтра. Как кратко упоминалось в главе 3, метод `join()` вычисляет, сколько кругов должно появиться или исчезнуть из диаграммы и какие из них должны быть обновлены. На основе этого можно создать интересный эффект, приложив совсем немного усилий.

Но есть проблема. Заметили, что при выборе фильтра круги не только появляются и исчезают на экране, но и перемещаются по диаграмме рассеяния? Большой оранжевый круг вверху (представляющий синих китов) внезапно становится меньше и меняет цвет на красный (соответствующий североатлантическому гладкому киту) и т. д. Нам всем нравится видеть движение на экране, но то, что происходит сейчас, никуда не годится. А происходит следующее: D3 автоматически обновляет количество кругов в соответствии с отфильтрованным набором данных. Затем назначает первый круг в DOM первому китообразному в обновленном наборе данных, второй круг – второму китообразному и т. д.

Для нас было бы желательно, чтобы круги, остающиеся на экране, продолжали представлять один и тот же вид китообразных. Этого можно добиться, передав в вызов метода `data()` функцию ключа во втором аргументе. С помощью этой функции мы сообщаем D3 сохранять данные, назначенные каждому элементу, оставшемуся на экране, как показано на рис. 7.18. В следующем фрагменте мы используем `uid` китообразных в качестве ориентира. Если вы знакомы с React, то этот прием похож на добавление ключевых атрибутов в элементы списка, которые обеспечивают им стабильную идентичность:

```
innerChart
  .selectAll("circle")
  .data(updatedData, d => d.uid)
  .join("circle")
    ...
```

Снова сохраните проект и поэкспериментируйте с фильтрами. Теперь диаграмма смотрится намного лучше!

В настоящее время при изменении фильтра круги, которые появляются на экране, въезжают в диаграмму слева, а круги, покидающие экран, просто исчезают. Можно ли управлять этими переходами? Мы упоминали, что при обновлении данных привязанные к диаграмме рассеяния круги (или данные, прикрепленные к ним)

могут появляться, обновляться или исчезать, как объясняется ниже и показано на рис. 7.19:

- *выборка появления* представляет новые элементы, которые необходимо создать. Например, если на диаграмме рассеяния в настоящее время выбран фильтр **Northern Hemisphere** (Обитающие в Северном полушарии) и выбирается фильтр **All Cetaceans** (Все китообразные), то на экране появятся новые круги;
- *выборка обновления* содержит элементы, которые уже присутствуют и останутся на экране. Эти элементы могут сохранять прежнее местоположение и стиль или изменяться, если привязанные к ним данные были обновлены. Например, если в настоящее время выбран фильтр **Northern Hemisphere** (Обитающие в Северном полушарии) и выбирается фильтр **All Cetaceans** (Все китообразные), то круги, представляющие китообразных из Северного полушария, должны остаться такими, какие они есть;
- *выборка исчезновения* представляет элементы, которые необходимо удалить. Например, если в настоящее время выбран фильтр **Northern Hemisphere** (Обитающие в Северном полушарии) и выбирается фильтр **Southern Hemisphere** (Обитающие в Южном полушарии), то круги, представляющие китообразных из Северного полушария, должны покинуть экран.

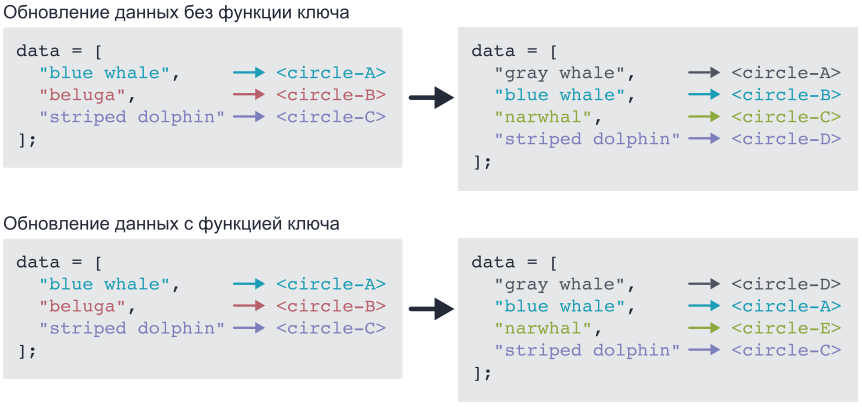


Рис. 7.18 Без функции ключа во время обновления данных D3 назначает первый элемент круга первому китообразному в обновленном наборе данных, второй элемент круга – второму китообразному и т. д. С функцией ключа элементы, остающиеся на экране, сохраняют те же данные

Метод `join()` облегчает нам жизнь, обрабатывая выборки появления, обновления и исчезновения, но мы все равно можем получить к ним доступ и управлять ими по отдельности. В следующем

фрагменте вместо передачи строки в вызов метода `join()`, как это делалось до сих пор, мы передаем отдельные функции для обработки выборок появления, обновления и исчезновения:

```
innerChart
  .selectAll("circle")
  .data(updatedData, d => d.uid)
  .join(
    enter => enter,
    update => update,
    exit => exit
  )
```

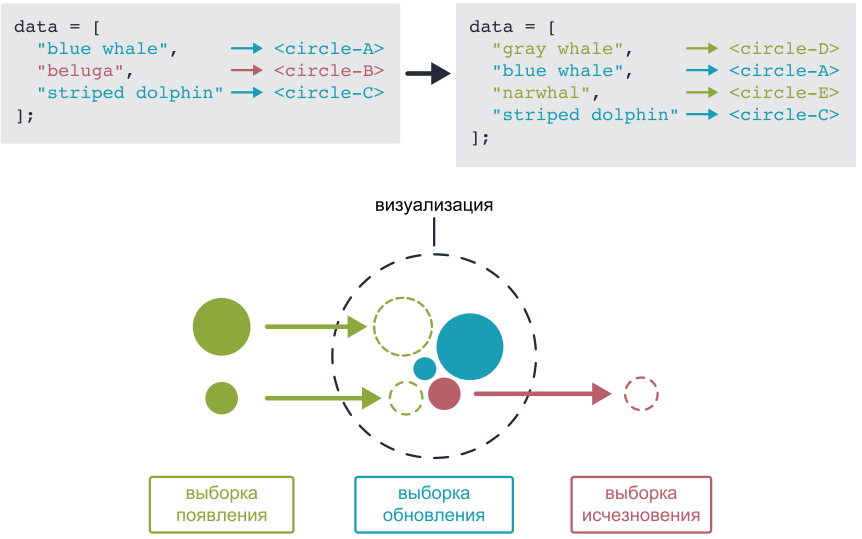


Рис. 7.19 Выборка появления представляет новые элементы, которые необходимо создать. Выборка обновления содержит элементы, которые уже отображаются и останутся на экране. Выборка исчезновения включает элементы, которые необходимо удалить из DOM

В листинге 7.13 мы указываем, что должно происходить с выборками появления, обновления и исчезновения. Когда новые данные добавляются в диаграмму рассеяния (выборка появления), мы добавляем соответствующее количество кругов и задаем их атрибуты. Здесь мы создаем анимацию, в которой круги появляются сверху, их радиусы плавно увеличиваются от 0 до конечного значения, а непрозрачность изменяется от 0 % до 100 %. Для этого мы сначала присваиваем атрибуту `cy` значение `-50`, атрибуту `r` значение `0` и свойству стиля `opacity` значение `0`. Затем используем метод `call()` для создания перехода и присваиваем атрибутам `cy`, `r` и `opacity` их конечные значения. Обратите внимание, что мы не включаем вызов `transition` в цепочку, как делали это до сих пор, а вызываем его внутри ме-

тода `call()`. Хотя нам не нужно применять никаких изменений к выборке обновления, мы все равно должны включить функцию `update` в вызов `join()`, чтобы обеспечить правильную работу этого метода.

Для обработки выборки исчезновения вызывается переход, перемещающий исчезающие круги вниз за пределы внутренней диаграммы с уменьшением их радиуса и непрозрачности до 0. В заключение мы удаляем выборку исчезновения из DOM, добавляя метод `remove()` в цепочку внутри `call()`.

Листинг 7.13 Выборки появления, обновления и исчезновения (interactions.js)

```
innerChart
  .selectAll("circle")
  .data(updatedData, d => d.uid)
  .join(
    enter => enter
      .append("circle")
        .attr("class", "cetacean")
        .attr("cx", d => xScale(
          d.global_population_estimate))
        .attr("cy", d => -50)
        .attr("r", 0)
        .attr("fill", d => colorScale(d.status))
        .attr('fill-opacity', 0.6)
        .attr("stroke", d => colorScale(d.status))
        .attr("stroke-width", 2)
        .style('opacity', 0)
      .call(enter => enter.transition()
        .attr("cy", d => yScale(d.max_size_m))
        .attr("r", d => rScale(d.max_weight_t))
        .style('opacity', 1)),
    update => update,
    exit => exit
      .call(exit => exit.transition()
        .attr("cy", d => innerHeight)
        .attr("r", 0)
        .style('opacity', 0)
        .remove())
```

Передать в join() функцию, обрабатывающую выборку появления

Добавить круг для каждой точки данных в выборке появления и присвоить атрибутам кругов начальные значения. Установить их вертикальное местоположение на 50 пикселей выше внутренней диаграммы, а радиус и непрозрачность равными 0

Передать в join() функцию, обрабатывающую выборку обновления, даже если с ней ничего не нужно делать

Добавить в цепочку вызов метода call(). Внутри него применить переход к выборке появления и задать конечные значения атрибутов и стилей: конечное местоположение по вертикали, радиус и непрозрачность

Передать в join() функцию, обрабатывающую выборку исчезновения

Добавить в цепочку вызов метода call(). Внутри него применить переход к выборке исчезновения и задать конечные значения атрибутов и стилей: конечное местоположение по вертикали, радиус и непрозрачность. Добавить также в цепочку вызов метода remove(), чтобы удалить эту выборку из DOM

7.4.3 Создание переходов для повторного использования

Переход по умолчанию, применяемый к выборкам появления и исчезновения, в целом неплох, но нам хотелось бы замедлить его и изменить функцию динамики. Если использовать ту же длительность и функцию динамики для анимации появления и исчезновения, то можно создать функцию перехода, которую можно будет использовать многократно.

В листинге 7.14 мы объявляем переход с помощью метода `d3.transition()`. Добавляем в цепочку метод `duration()` и устанавливаем длительность равной 800 мс. Также добавляем метод `ease()` и устанавливаем функцию динамики `d3.easeExpOut`. Полученную функцию перехода мы сохраняем в константу с именем `t`.

Затем нам остается только передать эту функцию перехода в вызовы метода `transition()`, применяемые к выборкам появления и исчезновения. Такие многоразовые функции перехода помогают избежать повторения кода, упрощают его поддержку и обеспечивают согласованность параметров перехода во всех проектах, где они используются.

Листинг 7.14 Создание перехода для повторного использования (interactions.js)

```
const handleClickOnFilter = (data) => {

  d3.selectAll(".filter")
    .on("click", (e, datum) => {
      if (!datum.isActive) {

        ...

        const t = d3.transition()
          .duration(800)
          .ease(d3.easeExpOut);

        innerChart
          .selectAll("circle")
          .data(updatedData, d => d.uid)
          .join(
            enter => enter
              .append("circle")
                .attr("class", "cetacean")
                ...
```

Объявить функцию перехода с помощью d3.transition(). Установить длительность и функцию динамики и сохранить результат в константе с именем t

```
            .call(enter => enter.transition(t)
              ...,

          update => update,
          exit => exit
            .call(exit => exit.transition(t)
              ...)

        )

    }
  });
};
```

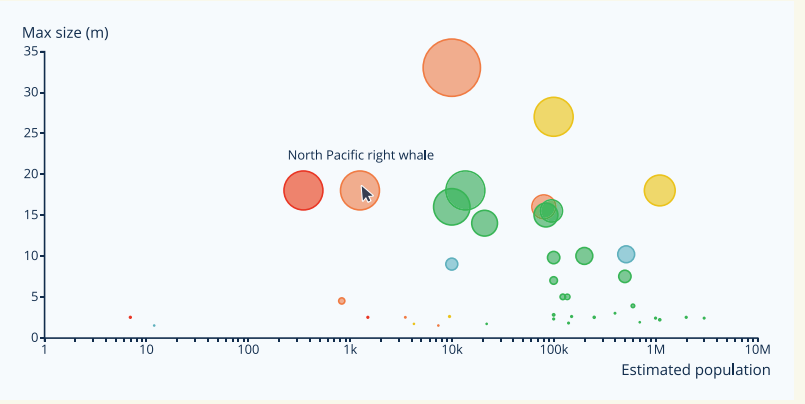
Передать функцию перехода t в существующие вызовы transition внутри метода join()

Упражнение: создание подсказки

У нас получилась очень привлекательная диаграмма рассеяния, но, глядя на нее, мы не можем сказать, какой вид китообразных представляет тот или иной круг. Чтобы исправить этот недостаток, выполните следующее:

1. добавьте простую подсказку, состоящую из одного текстового элемента SVG;
2. когда мышь оказывается над кругом, отобразите название вида китообразных;
3. когда мышь покидает пределы круга, скройте подсказку.

Пример вы можете увидеть на следующем изображении. (Обратите внимание, что обработчики событий нужно снова подключать к элементам, когда они появляются в DOM!)



При наведении указателя мыши на круг появляется подсказка

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в листингах D.6, D.7 и D.8, а также в папке `7.4-Refining_data_updates/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

Интервью с Мохамадом Вакедом

Мохамад Вакед (Mohamad Waked) – разработчик визуализаций и специалист по анализу данных.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

С удовольствием. Постараюсь рассказать свою длинную историю вкратце. Все началось в детстве. Я с детства увлекался исследованиями, рисованием, проектированием и строительством. Я испытывал особое чувство радости, когда, держа в руке карандаш, видел, как мои идеи и фантазии оживают на бумаге. Я чувствовал себя свободным и счастливым. Я также любил играть с числами и несомненно имел математические наклонности. Я мечтал стать инженером, чтобы нести в мир новые творческие идеи, и моя мечта сбылась. Я получил диплом инженера-механика и инженера-технолога. Свою карьеру я начал инженером по анализу процессов в автомобильной компании. Это была одна из моих первых встреч с данными. Ежедневно мне приходилось обрабатывать огромное количество данных и искать пути повышения производительности и эффективности производственной линии. Это заставило меня поступить на аспирантуру и заняться наукой о данных в Университете Джонса Хопкинса. В своей карьере я не раз отмечал, какой восторг испытываю, представляя результаты любого анализа в виде диаграммы или графика. В такие моменты я чувствовал, как высвобождается весь мой мозговой потенциал, потому что мне приходится думать и работать, используя всю мощь моего мозга – левую логическую и аналитическую половину вместе с правой творческой и художественной. Это стало основным стимулом моего второго перехода к визуализации данных. Я до сих пор помню тот день, когда услышал свой внутренний голос, говорящий: «Это моя жизненная задача. Здесь я могу выразить свою уникальность. Именно так я смогу служить другим».

После нескольких лет учебы и практики я решил создать Alhadaqa – мою личную лабораторию по визуализации данных. Там я смог предаться своей страсти и заняться визуализацией данных для людей через серию личных проектов на самые разные темы: серьезные, такие как мигранты или стихийные бедствия, и простые, такие как фильмы или спорт. Мои проекты завоевали различные награды на региональном и на мировом уровне, например Malofiej Awards, The Pudding Cup и золотую медаль Центра глобальной визуализации данных.

Что привело вас в D3? Сталкивались ли вы с трудностями при изучении этой библиотеки? Если да, то как вы их преодолевали?

За время своей карьеры я работал с различными инструментами создания диаграмм и графиков. Язык R с его интеллектуальной библиотекой ggplot2 дал мне возможность свободно создавать нестандартные диаграммы. Но когда я открыл для себя D3, то сразу понял, что это то, о чем я мечтал. Мне нравится разрабатывать диаграммы свободно, как будто

я рисую карандашом на бумаге. D3 дает мне эту высокую степень свободы в моих проектах, и не только в статической, но и в динамической форме.

Должен признать, что изучение D3.js было сложной задачей. Нелегко с ходу уловить логику, лежащую в основе всех строительных блоков библиотеки. Это требует времени, терпения и большой практики. Одна из основных стратегий, которые я использовал в ходе обучения, заключалась в исследовании устройства D3. Она состоит из модулей, которые для меня больше похожи на детали Lego™. Вы должны понять назначение каждой детали в отдельности, исследовать ее возможности, а затем попытаться объединить и смешать разные части вместе, чтобы понять, насколько далеко вы можете зайти в своем творчестве.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место занимает D3 в вашем рабочем процессе визуализации данных?

Я имею немалый опыт использования многих популярных инструментов визуализации. Но на этапе исследования данных, который может включать создание простых диаграмм и графиков, чтобы лучше понять данные, а также для быстрого тестирования некоторых начальных концепций дизайна я предпочитаю использовать R. Затем, на этапе построения визуализации данных, D3 становится для меня самым важным инструментом, и я активно использую его для тестирования и проверки осуществимости некоторых начальных концепций.

Представленный проект: «Migration between Search & Reality» (https://migrationinsearch.com/)



Проект проводит параллель между поисковыми запросами в Google, которые делают люди, отправляющиеся в миграцию, и реальностью их конечного пункта назначения

Опишите ваш проект «Migration between Search & Reality». Использование статистики поиска в Google для проведения параллелей между тем, что ищут мигранты, и реальностью иммиграции – это очень мощная концепция!

«Migration between Search & Reality» – это визуальное исследование, пытающееся количественно оценить, насколько велик разрыв между реальностью и мечтами мигрантов, интересами и желаниями, которые можно измерить с помощью их поисковых запросов. Это была попытка нарисовать целостную картину проблемы миграции во всем мире, чтобы лучше понять ее и связанные с ней проблемы, такие как пропажа или гибель мигрантов при попытке нелегального пересечения границ. Проект рассматривает проблему с двух сторон: стран эмиграции и иммиграции.

Спасибо за высокую оценку! Я рад, что вам и многим другим понравилась эта идея. Мы живем в век информации, и я считаю, что то, что когда-то было невозможным, сегодня стало возможным благодаря огромному объему данных, к которым мы имеем доступ и которые должны помочь нам лучше понимать и решать наши проблемы. Я также считаю, что специалисты по визуализации данных являются посланниками между миром данных и миром людей. Они могут говорить на языках обоих миров, и на них возложена очень серьезная роль служить мостом, соединяющим два мира.

Расскажите, как создавался этот проект.

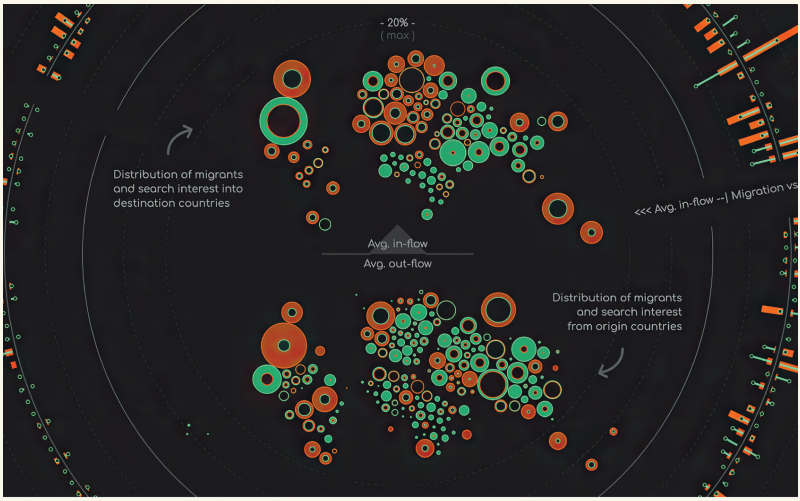
После согласования идеи и создания концептуального описания проекта я приступил к сбору данных о поисковых запросах в Google. Я благополучно справился с этой задачей, используя вариации типичных поисковых запросов, которые переводились на 10 самых распространенных языков мира, чтобы повысить точность результатов. Затем я смог бы сравнить эти данные о темах запросов с мировыми данными о миграции, полученными из международной миграционной базы данных ООН.

Подготовка, нормализация и анализ двух наборов данных тоже оказались непростой задачей. Могу честно сказать, что это был самый сложный проект по сбору и анализу данных, над которым я когда-либо работал. Зато по завершении этого этапа у меня сложилось четкое понимание самих данных и как эти данные можно визуализировать.

Далее были разработаны и опробованы некоторые первоначальные концепции дизайна. Они прошли целый ряд итераций, пока не достигли окончательного вида и не получили одобрение Альберто и Саймона [еще двух участников проекта]. Дизайн визуализации призван провести зрителя по кривой обучения от обычных столбчатых диаграмм и диаграмм с областями размаха к более сложным круговым интерактивным диаграммам «общей картины».

Вот один из моих маленьких секретов: за моими творческими дизайнерскими решениями всегда стоит вдохновение. В начале проекта мне может потребоваться время, чтобы найти это вдохновение, но после этого оно становится моим золотым дном. Оно остается рядом со мной на протяжении всего проекта, пока я продумываю различные дизайнерские решения.

При проведении исследований для этого проекта я нашел вдохновение в золотой медузе. Я был удивлен, что эти маленькие создания большую часть своей жизни проводят в ежедневной миграции, следуя за солнцем, движущимся по небу. Это перекликается с поведением мигрантов: они путешествуют, следуя за маленькой искрой света в конце пути. Если внимательно посмотреть на большую круговую диаграмму в конце истории, то можно заметить, насколько она похожа на большую круглую медузу. Две внутренние карты имитируют группы медуз, плавающих вместе. Кроме того, щелкнув на любой из стран, вы увидите четкий вид медузы спереди.



Основная визуализация – круги на карте мира

Создание простой сюжетной линии, которая бы уравновешивала объяснение и исследование данных, было сложной задачей. В качестве провоцирующего инцидента я выбрал сцену в аэропорту Кабула, когда США завершали вывод войск из Афганистана. Это было большое событие, о котором рассказывали в новостях по всему миру, поэтому многие могли соотнести его с собой. Оно также создало связь между Востоком и Западом в одном кадре. Затем я последовал модели Фрейтага, одной из моих любимых структур, и добавил настраиваемую читателем концовку.

Визуализация данных – это важнейшее ремесло со множеством ловушек, в которые легко может попасть каждый. Но одна из самых распространенных ловушек, на которую я хотел бы обратить внимание, – это мысль о том, что существует четкая грань между правильным и неправильным, и четкий набор правил, которым нужно следовать. Я считаю, что это в корне неверное представление, совершенно непригодное при работе с любыми проектами по визуализации данных, потому что все в этом мире относительно, и то, что может быть правильным в одном проекте, может быть неправильным в другом, и наоборот.

Мне нравится, что радиальная диаграмма в конце проекта позволяет читать визуализацию двумя способами: круги, сгруппированные как географическая карта в середине, и радиальная столбчатая диаграмма снаружи. Оба варианта предоставляют разный взгляд на данные. Было ли создание этой визуализации итеративным процессом или вы с самого начала четко знали, к чему идете?

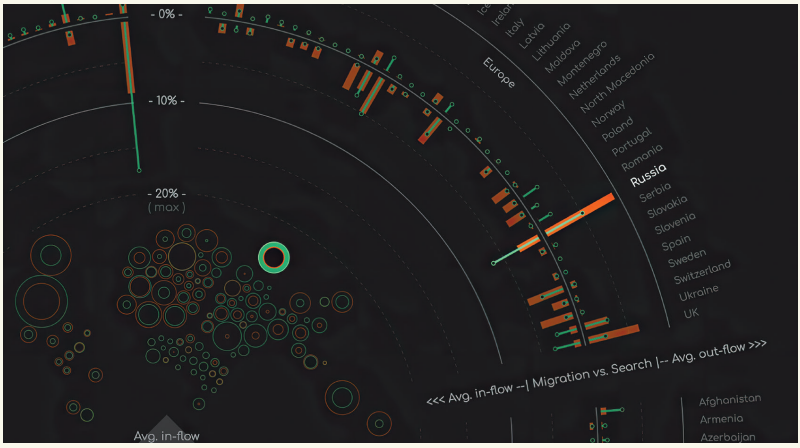
Спасибо большое. Я немного волновался о том, как люди воспримут эту сложную и нетрадиционную диаграмму, принимая во внимание, что другие диаграммы в проекте, предшествующие этой, были довольно простыми и распространенными. Но так получилось, что люди не смогли устоять перед желанием исследовать ее, судя по разным приятным отзывам, которые я получил после публикации.

В начале проекта у меня не было четкого представления о том, как все будет выглядеть, но я хотел создать что-то, что могло бы обобщить и представить полную картину этих двух параллельных миров (мечты и реальность мигранта) так, чтобы пользователи могли свободно исследовать их. На этапе изучения имеющихся данных в моем сознании подспудно стали развиваться некоторые концепции дизайна. Поэтому когда я подошел к этапу визуализации данных, эти базовые концепции переросли в зрелые концепции, которые можно было рассмотреть, зарисовать и протестировать.

Это также одна из особенностей моей личности: я во многом полагаюсь на свое подсознание в процессе создания визуализаций данных.

Есть еще кое-что, о чем я хотел бы упомянуть. Этот окончательный дизайн объединяет две основные концепции. Первая – маркированная столбчатая диаграмма, и вторая – диаграмма рассеяния пузырьков на двух объединенных географических картах. После глубоких размышлений я обнаружил, что объединение этих двух концепций даст больший эффект.

Этот проект поддержали Альберто Каиро (Alberto Cairo), Саймон Роджерс (Simon Rogers) и компания Google Trends. Расскажите немного о том, как вы пришли к сотрудничеству с ними.



Каждая страна отображается дважды: в радиальной столбчатой диаграмме и в диаграмме рассеяния в виде карты, что позволяет читателям рассматривать данные с двух точек зрения

Все началось с письма от Альберто Каиро, в котором он предложил создать проект для Google под его руководством. Единственное ограничение: проект должен основываться на данных поиска/тенденций Google. Разумеется, я сразу подумал, что это хорошая возможность поработать с такими признанными экспертами, как Альберто Каиро и Саймон Роджерс, и перенять частичку их опыта. Меня также воодушевляла свобода выбора темы. На самом деле все мои проекты связывают личное любопытство и свобода исследования.

После моей первой встречи с Альберто и Саймоном меня попросили предложить несколько идей. В конце концов мы согласились, что демонстрация разрыва между реальностью мигрантов в мире и их поисковыми запросами в Google оказалась самой интересной темой. Я был доволен выбором, поскольку он основывался на одном из моих предыдущих проектов – «The Unwelcomed» («Нежеланные») – о мигрантах и беженцах, погибающих при попытке пересечь границы государств.

Итак, отправной точкой стал выбор темы, а затем началось очень интересное путешествие исследований и дизайна.

Совсем недавно вы начали специализироваться на журналистике данных. Над какими проектами вы хотели бы работать в будущем?

Да, журналистика данных – интересная область, в которой я недавно начал работать, используя свои навыки и знания. Я рассматриваю это как личную и социальную ответственность, а не как работу. Что касается проектов, над которыми я хотел бы работать в будущем, то я всегда предпочитаю проекты, которые имеют смысл и могут иметь эффект, которые могут вызвать интерес к обсуждаемому вопросу, пролить свет на проблему или способствовать ее решению.

Итоги

- При планировании взаимодействий важно помнить о конечном пользователе. Всегда старайтесь организовывать такие взаимодействия, которые одновременно являются релевантными и интуитивными.
- В D3 обработчики событий устанавливаются с помощью метода `on()`. Этот метод принимает имя события в первом параметре (`click`, `mouseenter`, `mouseleave`, `mousemove` и т. д.) и функцию обратного вызова во втором параметре, которая будет получать имя события DOM и данные, прикрепленные к выборке.
- Мы можем управлять именами классов, присвоенных выбранному элементу, с помощью метода `classed()`. Этот метод принимает имя класса в первом параметре и логическое значение (`true` или `false`) во втором.
- Для создания плавной анимации переходов между состояниями используется метод `transition()`.
- Любой атрибут или стиль, включенный в цепочку после метода `transition()`, подвергается его воздействию.
- По умолчанию переходы в D3 длятся 250 мс, начинаются с задержкой 0 мс и используют функцию динамики `d3.easeCubicInOut`. Эти параметры можно изменить с помощью методов `duration()`, `delay()` и `ease()` соответственно.
- Переходы и их параметры можно объявлять отдельно и повторно использовать в проектах. Для этого функцию перехода нужно сохранить в константе и затем передать эту константу методу `transition()`.
- Для добавления аннотаций в диаграммы и отображения дополнительной информации можно использовать всплывающие подсказки. Создавать их можно с помощью элементов SVG. Их содержимое и местоположение можно устанавливать из функции обработки событий.
- Метод `d3.pointer()` принимает событие DOM и возвращает его координаты относительно цели. Это может пригодиться для позиционирования элементов на экране, когда требуется учитывать координаты события мыши.
- Логарифмическая шкала `d3.scaleLog()` возвращает логарифмическое преобразование непрерывной предметной области. Она особенно удобна, когда значения данных изменяются в очень широком диапазоне.
- Радиальная шкала `d3.scaleRadial()` применяется для отображения линейных значений в графическую метку, определяемую ее радиусом, например в окружность.

- При обновлении данных D3 автоматически обновляет элементы в DOM в соответствии с обновленным набором данных. Затем назначает первый элемент в DOM первому элементу данных в обновленном наборе данных, второй элемент – второму элементу данных и т. д. Чтобы данные «прилипали» к элементам, которые остаются на экране, нужно добавить ключевую функцию в вызов метода `data()`.
- Метод `join()` берет на себя заботу о выборках появления, обновления и исчезновения, но мы можем управлять ими, передавая соответствующие функции методу `join()` вместо строки.
- Выборка появления (`enter`) в методе `join()` представляет новые элементы, которые необходимо создать.
- Выборка обновления (`update`) в методе `join()` содержит элементы, которые уже присутствуют и останутся на экране.
- Выборка исчезновения (`exit`) в методе `join()` представляет элементы, которые необходимо удалить.

8 Интеграция D3 с фреймворками пользовательских интерфейсов

В этой главе рассматриваются:

- загрузка библиотеки D3 во фреймворки пользовательских интерфейсов;
- создание интерактивной панели с помощью D3 и React;
- манипуляция деревом DOM из D3 с использованием обработчиков React;
- использование D3 в качестве вспомогательной библиотеки и React для обновления и отображения DOM;
- применение тех же стратегий в Angular и Svelte.

До сих пор в этой книге мы создавали проекты простым и старомодным способом, используя только файлы HTML, CSS и JavaScript. Но современные проекты пользовательских интерфейсов обычно строятся с использованием фреймворков. Эти инструменты облегчают разработку крупных проектов и оптимизируют их производительность.

С помощью фреймворков JavaScript, таких как React, Angular, Svelte и Vue.js, можно создавать проекты, где все происходит мгновенно и обеспечивает ощущение мобильного приложения. Мы называем такие проекты одностраничными приложениями (Single-Page Applications, SPA), потому что с сервера загружается только один HTML-файл, который затем динамически обновляется.

Согласно опросу State of JavaScript 2021, наибольшей популярностью у разработчиков пользовательских интерфейсов пользуются такие фреймворки, как React (80 %), Angular (54 %), Vue.js (51 %) и Svelte (20 %). Философия и синтаксис этих инструментов могут сильно различаться, но использование D3 в сочетании с ними подчиняется ряду общих принципов. В этой главе мы обсудим эти принципы и применим их для создания интерактивной панели, изображенной на рис. 8.1 (онлайн-версия доступна по адресу http://mng.bz/yZlB). В этой панели мы воспроизведем визуализацию удовлетворенности разработчиков, интереса, использования и осведомленности о различных фреймворках пользовательских интерфейсов в период с 2016 по 2021 год, созданную командой, которая проводила опрос State of JavaScript 2021 (http://mng.bz/M9NW). Мы также построим диаграмму рассеяния процента лояльности (разработчиков, которые снова использовали бы фреймворк) в зависимости от количества пользователей для каждого фреймворка, а также столбчатую диаграмму их осведомленности о каждом инструменте.

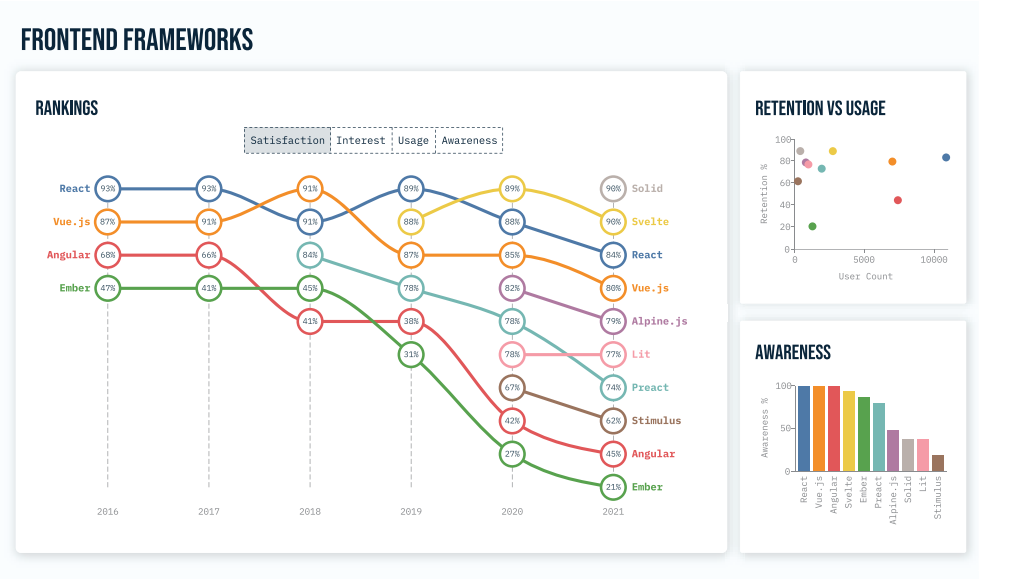


Рис. 8.1 Информационная панель с результатами опроса разработчиков пользовательских интерфейсов, которую мы создадим в этой главе

Мы построим эту панель, интегрировав D3 в приложение React. React – это золотой стандарт и, безусловно, самый популярный фреймворк. Но поскольку у вас может появиться желание использовать D3 также с другими фреймворками, мы предоставим дополнительную информацию об интеграции D3 с Angular и Svelte. В файлах с примерами кода для этой главы (http://mng.bz/amW7) вы найде-

те реализацию панели с помощью этих трех фреймворков. Позже, в главах 14 и 15, мы создадим еще один проект с D3 и Svelte.

> **ПРИМЕЧАНИЕ** Технически React является библиотекой, а Svelte – компилятором, а не фреймворком. Но для простоты мы будем считать, что React и Svelte – это *фреймворки*.

Поскольку основное внимание в этой книге уделяется библиотеке D3, мы рекомендуем вам получить хотя бы базовые знания о React, прежде чем читать эту главу. Вы должны знать, как создавать компоненты, передавать входные свойства дочерним и родительским компонентам, управлять состоянием компонентов и использовать обработчики React. Если вы впервые слышите о React, то вот несколько ресурсов, которые могут помочь вам начать:

- *React Quickly, Second Edition*, Мортена Барклунда (Morten Barklund) и Азата Мардана (Azat Mardan) (Manning, 2023)[1];
- видеокурс *React.JS: The Complete Course for Beginners* (видеоурок), Meta Brains (Manning, 2022)[2];
- курс Максимилиана Шварцмюллера (Maximilian Schwarzmüller) *React: The Complete Guide* (Safari, 2021)[3].

8.1 Подходы к использованию D3 во фреймворках пользовательских интерфейсов

Основная задача фреймворка пользовательского интерфейса – управление деревом DOM, исходя из состояния приложения. Фреймворки следят за тем, какие части DOM необходимо отобразить или обновить в каждый конкретный момент времени. Библиотека D3 была создана до появления таких инструментов и поэтому тоже включает средства управления DOM. Если попытаться использовать D3 так, как мы делали это до сих пор, в интерфейсе, построенном на основе фреймворка, то это может привести к конфликтам, когда фреймворк и D3 начнут манипулировать одними и теми же элементами. По этой причине нужно проявлять осторожность и использовать один из следующих подходов:

---

[1] *Азат Мардан, Мортен Барклунд*. React быстро. СПб.: Питер, 2021. ISBN: 978-5-4461-2220-2. – *Прим. перев.*

[2] https://www.manning.com/livevideo/react.js-the-complete-course-for-beginners%22.

[3] https://www.udemy.com/course/react-the-complete-guide-incl-redux/?couponCode=D_1122%22.

- предоставить библиотеке D3 контроль над частью DOM;
- использовать D3 как вспомогательную библиотеку;
- применить гибридный подход.

Первый подход заключается в том, чтобы изолировать элемент DOM и передать управление им библиотеке D3. Главное преимущество этого подхода – простота. Он позволяет использовать D3 так же, как мы делали это до сих пор. Но многие считают такой подход слишком грубым, ведущим к потере некоторых функций оптимизации, предлагаемых фреймворками JavaScript.

Второй подход является полной противоположностью. Суть его заключается в том, чтобы возложить ответственность за управление деревом DOM на фреймворк, а библиотеку D3 использовать только как сборник вспомогательных функций. Эта стратегия подразумевает отказ от шаблона привязки данных, генераторов осей, управления атрибутами и стилями, обработки событий и организации переходов, но оставляет доступными многие мощные методы D3, такие как шкалы и генераторы форм. Этот подход позволяет создавать все визуализации, обсуждавшиеся до сих пор, пусть и ценой некоторых дополнительных усилий, зато поддерживает оптимизации, обеспечиваемые фреймворками.

При гибридном подходе мы передаем управление деревом DOM фреймворку в максимально возможной степени, но оставляем ограниченный контроль за D3 там, где это даст явный выигрыш во времени разработки или где обходной путь слишком сложен. Например, в главе 7 мы обсудили, насколько эффективны и просты в использовании переходы D3, но они требуют, чтобы D3 манипулировала группами элементов DOM. В этом случае мы должны оценить имеющиеся у нас возможности. Для создания переходов, основанных исключительно на изменении свойств CSS, наилучшим решением является использование анимаций и переходов CSS или даже сторонней библиотеки React, такой как react-spring. Но если переход основан на изменении атрибута `d` элемента `path`, то переходы D3 являются, пожалуй, самым простым и эффективным способом. Другим примером может служить метод `brush()` из библиотеки D3, который тоже манипулирует элементами DOM.

В этой главе мы продемонстрируем все три подхода на примере создания информационной панели, как показано на рис. 8.2. Сначала мы построим диаграмму рассеяния процента лояльности к фреймворкам, используя первый подход. Здесь мы позволим библиотеке D3 управлять частью DOM. Затем перестроим эту диаграмму, используя D3 как вспомогательную библиотеку и возложив управление деревом DOM исключительно на React. Создание той же диаграммы с применением двух подходов позволит вам оценить плюсы и минусы каждого из них.

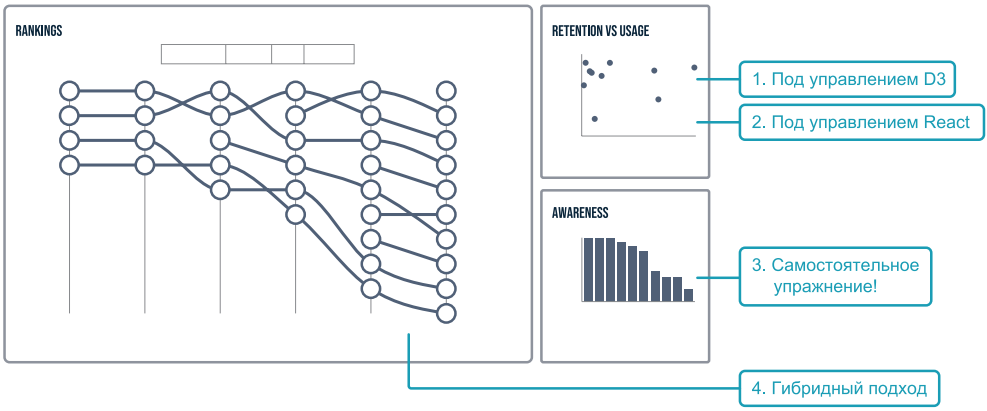


Рис. 8.2 Мы построим информационную панель с использованием трех подходов, описанных выше. Сначала создадим диаграмму рассеяния, применив первую стратегию и предоставив библиотеке D3 доступ к дереву DOM. Затем перестроим эту диаграмму с помощью React, используя D3 только как вспомогательную библиотеку. Потом вам будет предложено самостоятельно построить столбчатую диаграмму, используя второй подход. Наконец, мы построим визуализацию рейтингов, используя гибридную стратегию – сама диаграмма будет построена с помощью React, но при этом библиотеке D3 будет позволено изменять часть DOM для реализации плавного перехода кривых, когда пользователь щелкает на одной из кнопок

Затем вам будет предложено самостоятельно построить столбчатую диаграмму осведомленности разработчиков о фреймворках, используя второй подход. Это упражнение поможет вам закрепить вновь полученные знания.

Наконец, мы построим интерактивную визуализацию рейтингов, используя гибридную стратегию. Саму диаграмму мы построим с помощью второго подхода, но позволим D3 управлять деревом DOM для реализации плавного перехода кривых, когда пользователь щелкает на одной из кнопок над визуализацией.

8.2 Установка библиотеки D3 в проект React

Чтобы реализовать проект, представленный в этой главе, загрузите файлы кода из репозитория книги на GitHub (http://mng.bz/Xqjv). Откройте папку `start`, находящуюся в папке `chapter_08/8.2-Installing_D3`, в редакторе.

> **ПРИМЕЧАНИЕ** Папка `chapter_08` содержит несколько подпапок с именами, соответствующими названиям разделов главы. Каждая из подпапок содержит папки `start` и `end`. В папке находится код, соответствующий началу раздела, а в папке `end` содержится уже готовое решение.

Если у вас есть опыт работы с фреймворками пользовательского интерфейса, то вы наверняка знакомы с Node Package Manager (NPM), реестром программного обеспечения, откуда мы устанавливаем библиотеки и инструменты в наши проекты. Чтобы установить зависимости для нашего проекта, выполните следующую команду в терминале. Не забудьте в терминале перейти в папку проекта. Если вы используете терминал, интегрированный в Visual Studio Code (VS Code), то текущим каталогом автоматически будет назначен корневой каталог проекта:

```
npm install
```

Чтобы добавить библиотеку D3 в проект, установите пакет `d3` (www.npmjs.com/package/d3):

```
npm install d3
```

Наконец, запустите проект React командой

```
npm run start
```

Проект должен автоматически открыться в браузере по умолчанию. Вы также можете открыть его, введя в браузере адрес `http://localhost:3000`. Мы рекомендуем использовать Chrome или Firefox, так как эти браузеры имеют удобные инструменты разработчика. В настоящий момент вы должны увидеть в окне браузера только надпись «Loading...» (Загрузка...), потому что в проект пока не загружено никаких данных. Мы сделаем это в следующем разделе.

Одно из ключевых преимуществ фреймворков – они побуждают нас разбивать код на небольшие компоненты. В папке `src` проекта для этой главы компоненты информационной панели разбиты на четыре подпапки:

- `ChartComponents` – отдельные элементы диаграммы, такие как круги, прямоугольники и оси;
- `Charts` – полные диаграммы, такие как диаграмма рассеяния или столбчатая диаграмма;
- `Interactions` – бóльшая часть кода, отвечающего за взаимодействия;
- `UI` – элементы пользовательского интерфейса, не связанные с диаграммами, такие как кнопки и карточки.

Прежде чем начать использовать D3, импортируем ее, чтобы получить возможность использовать ее для создания компонентов диаграммы. Для этого откройте файл `App.js` и в самом верху добавьте инструкцию импорта:

```
import * as d3 from 'd3';
```

Инструкции, подобные предыдущей, мы называем импортом пространства имен. Здесь `d3` – это объект пространства имен, содержащий все сущности, экспортируемые из библиотеки D3. После добавления этой инструкции мы сможем обратиться к любому методу D3 с помощью точечной нотации, как мы делали до сих пор, например `d3.scaleLinear()` и `d3.line()`. Библиотеку D3 нужно импортировать в каждый файл, где она будет использоваться.

> **СОВЕТ** Вместо загрузки всей библиотеки D3 мы могли бы установить только те модули D3, которые используются в проекте, например `d3-scale` или `d3-shape`. Именно так мы и начнем поступать в главе 11.

В проекте React подавляющее большинство файлов находятся внутри папки `src`, где `index.js` является главным файлом и загружает `App.js`. Именно в `App.js` мы начинаем писать наш код и загружать компоненты.

8.3 Загрузка данных в проект React

Если внимательно рассмотреть `App.js`, то можно заметить, что он состоит из функционального компонента. В терминологии React функциональным компонентом называют функцию JavaScript, которая принимает аргументы, называемые *props* (входные свойства), и возвращает разметку компонента в формате JavaScript XML (JSX). В этой главе мы сосредоточимся на функциональных компонентах, поскольку это самый современный способ создания компонентов React, широко используемый в сообществе React. Однако старый способ создания компонентов, называемых компонентами на основе классов, все еще актуален, и в интернете вы найдете много примеров использования D3 в компонентах на основе классов.

Компонент `App` имеет переменную состояния с именем `loading`, которая инициализируется значением `true`, как показано в листинге 8.1. Он возвращает элемент `div`, содержащий условие. Если `loading` имеет значение `true`, то на экран выводится надпись `"Loading..."`. Если `loading` имеет значение `false`, то загружается компонент с именем `Charts`, который позже будет содержать три диаграммы.

Листинг 8.1 Начальный код в App.js

```
import { useState } from 'react';
import * as d3 from 'd3';
import Charts from './Charts/Charts';

const App = () => {
```

Импортировать ловушку React State, библиотеку D3 и компонент Charts

```
  const [loading, setLoading] = useState(true);
  return (
    <div className="container">
      {loading && <div className="loading">
        Loading...</div>}
      {!loading && <Charts />}
    </div>
  );
};

export default App;
```

Объявить переменную состояния с именем "loading" и инициализировать ее значением true

Если loading имеет значение true, то вернуть div с текстом «Loading...». Если loading имеет значение false, то вернуть компонент Charts

Мы структурировали компонент App именно так, потому что мы должны загрузить данные в приложение, прежде чем отображать диаграммы. А поскольку извлечение данных является асинхронным процессом, мы должны дождаться, пока данные станут доступны.

В функциональных компонентах ловушка эффектов служит точкой получения данных. В листинге 8.2 мы импортируем ловушку `useEffect` из библиотеки React. Затем вызываем ее и передаем ей анонимную функцию в первом аргументе. В этой функции мы загружаем JSON-данные из REST API по адресу http://mng.bz/g7ZV с помощью метода `d3.json()`. До сих пор мы загружали данные только из CSV-файлов, входящих в состав проекта, но в профессиональных проектах данные чаще извлекаются из REST API.

Вслед за методом `d3.json()` вызывается JavaScript-метод `then()`, который возвращает объект `Promise`. Как только данные станут доступны, мы вызываем функции `setData()` и `setLoading()` для обновления состояния компонента. Обратите внимание, что мы также объявили переменную состояния `data` и присвоили ей пустой массив как начальное значение.

Второй аргумент `useEffect()` – это зависимости эффекта, которые сообщают библиотеке React, когда эффект должен выполняться. Поскольку под видом зависимостей мы передали пустой массив, то функция будет вызвана только один раз, сразу после монтирования компонента. По завершении мы передаем обновленную переменную состояния данных компоненту `Charts` как входное свойство.

Листинг 8.2 Загрузка данных в App.js

```
import { useState, useEffect } from 'react';
import * as d3 from 'd3';
import Charts from './Charts/Charts';

const App = () => {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState([]);
```

Импортировать из библиотеки React ловушку эффекта

Объявить переменную состояния с именем data и присвоить ей пустой массив как начальное значение

```
useEffect(() => {

  const dataURL = "https://d3js-in-action-third-edition.github.io/
    hosted-data/apis/front_end_frameworks.json";

  d3.json(dataURL).then(data => {
    setData(data);
    setLoading(false);
  });

}, []);

return (
  <div className="container">
    {loading && <div className="loading">Loading...</div>}
    {!loading && <Charts data={data} />}
  </div>
);
};

export default App;
```

С помощью метода d3.json() извлечь данные из API, адрес которого хранится в константе dataURL. Как только данные станут доступны (возвращены методом then()), записать загруженные данные в переменную состояния data и присвоить переменной состояния loading значение false

Передать переменную состояния data в качестве входных данных в компонент Charts

Передать в useEffect() пустой массив во втором аргументе. Это гарантирует запуск функции только один раз и сразу после монтирования компонента

Функция очистки useEffect()

В данном проекте это маловероятно, но вообще пользователь может потерять терпение и перейти на другую страницу до того, как D3 завершит процесс извлечения данных. В этом случае React отмонтирует компонент, извлекающий данные. Если запрос все еще не успел выполниться, то преждевременное демонтирование может привести к нежелательному поведению, например к утечкам памяти.

Чтобы избежать таких проблем, можно добавить функцию очистки в ловушку `useEffect`. Это возвращаемая функция, как показано в следующем примере. Если компонент будет отмонтирован до завершения извлечения данных, то React запустит функцию очистки.

Есть много разных способов обработки таких ситуаций, но этот самый простой. В примере ниже внутри ловушки `useEffect` объявляется переменная `mounted` и ей присваивается начальное значение `true`. В функции очистки, которая будет вызвана при отмонтировании компонента, переменной `mounted` присваивается `false`. Наконец, внутрь метода `then()` добавлено условие – переменные состояния обновляются только в том случае, если `mounted` имеет значение `true`. Таким образом отмонтированный компонент не будет пытаться обновить переменные состояния и тем самым избежит ошибок:

```
useEffect(() => {
  ...

  let mounted = true
  d3.json(dataURL).then(data => {
    if (mounted) {
      setData(data);
      setLoading(false);
    }
  });

  return () => mounted = false;
}, []);
```

Если внутри метода `then()` вывести переменную `data` в консоль, то можно заметить, что она состоит из трех массивов. Массив `ids` содержит список фреймворков, включенных в набор данных, а массив `years` перечисляет годы, по которым есть данные. Массив `experience` содержит объект для каждого фреймворка с его идентификатором, именем, количеством пользователей (согласно данным опроса, State of JavaScript Survey 2021) и процентами лояльности, а также массивами в атрибутах `satisfaction`, `interest`, `usage` и `awareness`, которые представляют по годам рейтинг фреймворка и процент респондентов, ответивших, что они удовлетворены, заинтересованы, используют или знают о фреймворке. Теперь данные готовы к использованию!

> **ПРИМЕЧАНИЕ** Возможно, вы заметите, что данные дважды выводятся в консоль, хотя мы говорили, что анонимная функция внутри ловушки эффекта будет вызвана только один раз. Это преднамеренная особенность React, называемая `StrictMode`. Ее цель – помочь обнаружить случайные побочные эффекты при отображении интерфейса приложения. Такое происходит только в режиме разработки.

8.4 Многоразовый подход к контейнерам SVG

В главе 4 этой книги мы познакомились с подходом к построению диаграмм D3, который включает контейнер SVG и элемент группы, как показано на рис. 8.3. Эта группа смещается в нужное положение при помощи отступов и вмещает все элементы, составляющие внутреннюю диаграмму. Поскольку элементы внутри группы SVG наследуют свойства своего родителя, все они смещаются вместе с группой.

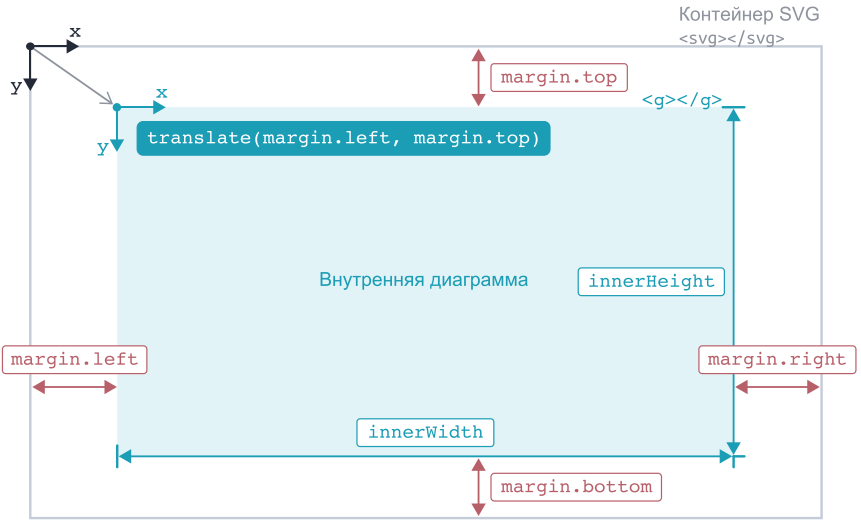


Рис. 8.3 Стратегия, которую мы использовали начиная с главы 4, заключается в построении диаграмм внутри контейнера SVG, в который добавляется элемент группы. Эта группа позиционируется при помощи отступов и вмещает все элементы, составляющие внутреннюю диаграмму. Элементы внутри группы SVG наследуют свойства своего родителя и смещаются вместе с группой

Наша информационная панель будет содержать три диаграммы, поэтому было бы полезно создать повторно используемый компонент контейнера SVG, содержащий как родительский элемент SVG, так и группу, обертывающую внутреннюю диаграмму. Откройте файл `ChartContainer.js` в папке `ChartComponents`. Он содержит определение функционального компонента с именем `ChartContainer`, который получает входные свойства и возвращает элемент SVG.

Чтобы последовать стратегии, показанной на рис. 8.3, мы добавим элемент группы внутрь контейнера SVG. В листинге 8.3 мы предполагаем, что этот компонент получит три входных свойства: `width`, `height` и объект `margin`. Свойства `width` и `height` применяются к атрибуту `viewBox` элемента SVG. Затем мы устанавливаем атрибут `transform` группы и смещаем ее по горизонтали на величину отступа слева и по вертикали на величину отступа сверху.

Наконец, зная, что в React каждый компонент получает входное свойство с именем `children`, содержащее дочерние элементы, определенные в родительском компоненте, мы можем вернуть это свойство, заключив его в фигурные скобки: `{props.children}`.

Листинг 8.3 Создание ChartContainer (ChartContainer.js)

```
const ChartContainer = props => {
  return (
```

```
    <svg viewBox={`0 0 ${props.width} ${props.height}`}>
      <g transform={`translate(${props.margin.left},
        ${props.margin.top})`}>
        {props.children}
      </g>
    </svg>
  );
};

export default ChartContainer;
```

Настроить атрибут viewBox контейнера SVG с использованием параметров width и height

Переместить группу SVG на величину отступов слева и сверху

Вернуть дочерние элементы компонента

Затем откройте файл `Charts.js`, содержащийся в папке `Charts`, и определите объект `margin`. В листинге 8.4 этот объект передается как входное свойство трем компонентам диаграммы (`Rankings`, `ScatterplotD3Controlled` и `BarChart`).

Листинг 8.4 Передача объекта margin как входного свойства (Charts.js)

```
const Charts = props => {
  const margin = {top: 30, right: 10, bottom: 50,
    left: 60};

  return (
    <Fragment>
      <h1>Front-end Frameworks</h1>
      <div className='row'>
        <div className='col-9'>
          <Rankings margin={margin} />
        </div>
        <div className='col-3'>
          <div className='row'>
            <div className='col-12'>
              <ScatterplotD3Controlled margin={
                margin} />
            </div>
            <div className='col-12'>
              <BarChart margin={margin} />
            </div>
          </div>
        </div>
      </div>
        ...
    </Fragment>
  )
};
```

Объявить объект margin

Передать margin как входное свойство компонентам Rankings, ScatterplotD3Controlled и BarChart

Чтобы добавить контейнер диаграммы, выполните следующие действия:

1. перейдите к компоненту `ScatterplotD3Controlled`, расположенному в папке `Charts`;
2. в верхней части файла импортируйте компонент `ChartContainer`;
3. внутри функции компонента объявите константы `width` и `height` со значениями `300` и `245` соответственно;
4. рассчитайте `innerWidth` и `innerHeight` диаграммы, вычтя отступы, полученные в виде входных свойств, из размеров, как показано в листинге 8.5;
5. непосредственно перед закрывающим тегом карточки (`</Card>`) вызовите компонент `ChartContainer` и передайте ему входные свойства `width`, `height` и `margin`.

Листинг 8.5 Вызов ChartContainer (ScatterplotD3Controlled.js)

```
import Card from '../UI/Card';
import ChartContainer from '../ChartComponents/
  ChartContainer';

const ScatterplotD3Controlled = props => {
  const width = 300;
  const height = 245;
  const innerWidth = width - props.margin.left -
    props.margin.right;
  const innerHeight = height - props.margin.top -
    props.margin.bottom;

  return (
    <Card>
      <h2>Retention vs Usage</h2>
      <ChartContainer
        width={width}
        height={height}
        margin={props.margin}
      >
      </ChartContainer>
    </Card>
  )
};

export default ScatterplotD3Controlled;
```

Импортировать компонент ChartContainer

Объявить константы width и height. Рассчитать innerWidth и innerHeight диаграммы, вычитая отступы из размеров

Добавить компонент ChartContainer. Передать ему width, height и margin. Этот компонент имеет открывающий и закрывающий теги, между которыми мы позже добавим диаграмму

Проделаем то же самое с компонентами `BarChart` и `Rankings`. Размеры столбчатой диаграммы такие же, как и размеры диаграммы рассеяния, а диаграмма рейтингов имеет ширину 1000 пикселей и высоту 542 пикселя. Также увеличим левый и правый отступы диаграммы рейтингов, как показано в следующем фрагменте кода:

```
const width = 1000;
const height = 542;
const marginRight = 150;
const marginLeft = 110;
const innerWidth = width - props.marginLeft - props.marginRight;
const innerHeight = height - props.margin.top - props.margin.bottom;
```

Сохраните все измененные файлы и откройте проект в браузере. Взглянув на разметку в инспекторе, вы увидите, что для каждой диаграммы был добавлен контейнер SVG и элемент группы, а их атрибуты получили ожидаемые значения. Так мы создаем повторно используемые компоненты SVG в React!

8.5 Передача части DOM под управление D3

Теперь мы готовы нарисовать первую визуализацию, используя первый подход, как показано на рис. 8.4. В этом разделе мы построим диаграмму рассеяния, которую можно увидеть на рис. 8.1 и в онлайн-версии проекта (http://mng.bz/yZlB), позволив D3 управлять частью DOM.

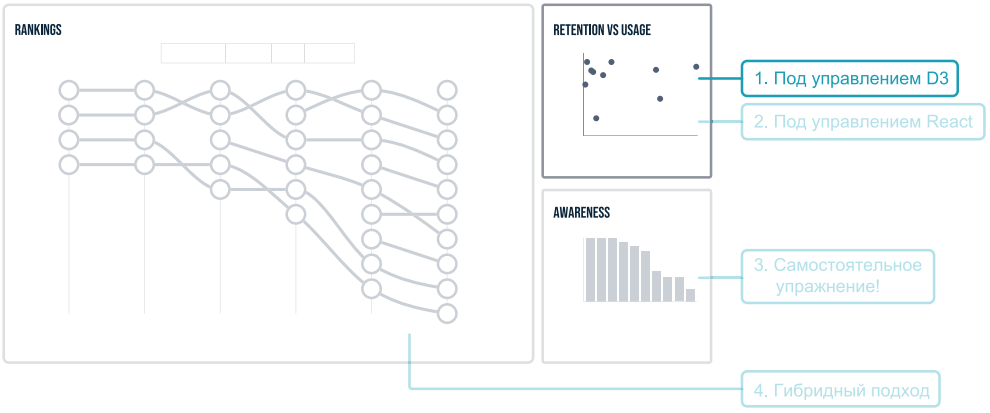


Рис. 8.4 В этом разделе мы построим диаграмму рассеяния, иллюстрирующую зависимость лояльности пользователей от их количества, позволив D3 управлять частью DOM

8.5.1 React

В React управление элементом передается по ссылке с помощью ловушки `useRef()`, как подробно показано на рис. 8.5.

Сначала мы должны импортировать ловушки `useRef` и `useEffect` из React. Внутри компонента инициализировать `useRef` и сохранить результат в константе, например `myRef`. Затем присвоить атрибуту

`ref` элемент, управление которым предполагается передать библиотеке D3, и указать константу, в которой была сохранена ловушка (`myRef`).



Рис. 8.5 Чтобы позволить D3 управлять частью DOM в React, сначала нужно импортировать ловушки `useRef()` и `useEffect()` в компонент. Затем инициализировать ссылку и сохранить ее как атрибут элемента, которым будет управлять D3. Наконец, внутри ловушки `useEffect()` нужно выбрать текущий экземпляр по ссылке – и можно начинать использовать D3 как обычно

Второй шаг – использование ловушки эффекта для создания функции, в которой можно использовать D3. В анонимной функции, которая передается в `useEffect` в первом аргументе, нужно с помощью D3 выбрать текущую версию ссылки (`d3.select(myRef.current)`). Свойство `current` гарантирует актуальность элемента, которым будет манипулировать библиотека. С этого момента можно использовать D3 в точности, как мы делали это раньше, но только в рамках анонимной функции `useEffect`.

Чтобы применить эту стратегию к диаграмме рассеяния, откройте файл `ScatterplotD3Controlled.js` в папке `Charts`. В самом начале импортируйте ловушки `useRef` и `useEffect` из React, а также библиотеку D3. Внутри компонента инициализируйте ловушку `useRef` и сохраните ее в константе с именем `scatterplotRef`, как показано в листинге 8.6. Внутрь `ChartContainer` добавьте элемент группы и присвойте

его атрибуту `ref` ссылку на `scatterplotRef`. Наконец, создайте ловушку `useEffect` и используйте D3 для выбора текущего свойства `ref`.

Листинг 8.6 Подход с передачей управления частью DOM библиотеке D3 (ScatterplotD3Controlled.js)

```
import { useRef, useEffect } from 'react';
import * as d3 from 'd3';

const ScatterplotD3Controlled = props => {
  ...

  const scatterplotRef = useRef();
  useEffect(() => {
    const scatterplotContainer = d3.select(
      scatterplotRef.current);
  }, []);

  return (
    <Card>
      <h2>Retention vs Usage</h2>
      <ChartContainer
        ...
      >
        <g ref={scatterplotRef}></g>
      </ChartContainer>
    </Card>
  )
};

export default ScatterplotD3Controlled;
```

Импортировать ловушки useRef и useEffect из React. Импортировать библиотеку D3

Инициализировать ловушку useRef и сохранить ее в константе с именем scatterplotRef

Внутри анонимной функции для ловушки useEffect выбрать с помощью D3 текущее свойство ref

Добавить элемент группы внутрь ChartContainer и присвоить его атрибуту ref ссылку на scatterplotRef

Прежде чем продолжить строить диаграмму рассеяния, вернемся к компоненту `Charts` и объявим цветовую шкалу. Для этого мы выбрали порядковую шкалу D3, которая отображает дискретную предметную область в дискретный диапазон. Предметная область в данном случае – это массив идентификаторов фреймворков, доступных в данных, а в роли диапазона мы используем одну из предопределенных цветовых палитр D3, `d3.schemeTableau10`, доступную в модуле d3-scale-chromatic:

```
import * as d3 from 'd3';

const colorScale = d3.scaleOrdinal()
  .domain(props.data.ids)
  .range(d3.schemeTableau10);
```

Передадим компоненту `ScatterplotD3Controlled` цветовую шкалу и массив `experience` как входные свойства `colorScale` и `data` соответственно:

```
<ScatterplotD3Controlled
  margin={margin}
  data={props.data.experience}
  colorScale={colorScale}
/>
```

Теперь можно продолжить строить диаграмму рассеяния с помощью D3. По горизонтальной оси мы будем откладывать количество пользователей, а по вертикальной – процент лояльности. Вернемся к `ScatterplotD3Controlled` и внутри анонимной функции ловушки `useEffect` объявим шкалы, сгенерируем оси и используем шаблон привязки данных, чтобы добавить элемент круга для каждого фреймворка, как показано в листинге 8.7. Если вы читали предыдущие главы, то эти шаги должны быть вам знакомы.

Обратите внимание, как передаются зависимости `innerWidth`, `innerHeight` и `props` в ловушку `useEffect`. Эти переменные используются внутри функции `useEffect`, и если не добавить их в массив `dependencies`, то React выдаст предупреждение. Но поскольку эти переменные не изменяются, функция `useEffect` по-прежнему будет запускаться только один раз.

Листинг 8.7 Построение диаграммы рассеяния (ScatterplotD3Controlled.js)

```
const ScatterplotD3Controlled = props => {
  ...

  const scatterplotRef = useRef();
  useEffect(() => {
    const scatterplotContainer = d3.select(scatterplotRef.current);

    const xScale = d3.scaleLinear()
      .domain([0, d3.max(props.data, d =>
        d.user_count)])
      .range([0, innerWidth])
      .nice();
    const yScale = d3.scaleLinear()
      .domain([0, 100])
      .range([innerHeight, 0]);
```

Объявить две линейные шкалы для горизонтальной и вертикальной осей. Предметная область горизонтальной оси простирается от 0 до максимального количества пользователей любого фреймворка. Предметная область вертикальной оси простирается от 0 до 100 и определяет процент лояльности

```
    const bottomAxis = d3.axisBottom(xScale)
      .ticks([3])
      .tickFormat(d3.format("d"));
    scatterplotContainer
      .append("g")
        .attr("class", "axis")
        .attr("transform", `translate(0,
          ${innerHeight})`)
        .call(bottomAxis);
    const leftAxis = d3.axisLeft(yScale)
      .ticks([5]);
    scatterplotContainer
      .append("g")
        .attr("class", "axis")
        .call(leftAxis);

    scatterplotContainer
      .selectAll(".scatterplot-circle")
      .data(props.data)
      .join("circle")
        .attr("class", "scatterplot-circle")
        .attr("cx", d => xScale(d.user_count))
        .attr("cy", d => yScale(d.retention_percentage))
        .attr("r", 6)
        .attr("fill", d => props.colorScale(d.id));
  }, [innerWidth, innerHeight, props]);

  return (
    <Card>
      <h2>Retention vs Usage</h2>
      <ChartContainer ... >
        <g ref={scatterplotRef}></g>
      </ChartContainer>
    </Card>
  )
};
```

Инициализировать генераторы осей и добавить оси во внутреннюю диаграмму

Использовать шаблон привязки данных, чтобы добавить по одному кругу для каждого фреймворка. Горизонтальное положение кругов определяется количеством пользователей, а вертикальное – процентом лояльности. Для настройки их атрибутов fill вызывается цветовая шкала, полученная как входное свойство, которой передается идентификатор фреймворка

Добавить требуемые зависимости. Поскольку эти переменные не изменяются, функция внутри ловушки useEffect будет вызвана только один раз

По завершении диаграмма рассеяния должна выглядеть, как показано на рис. 8.6. Обратите внимание, что сейчас мы не будем тратить время на стилизацию осей и присваивание им меток, но вы можете сделать это, если хотите.

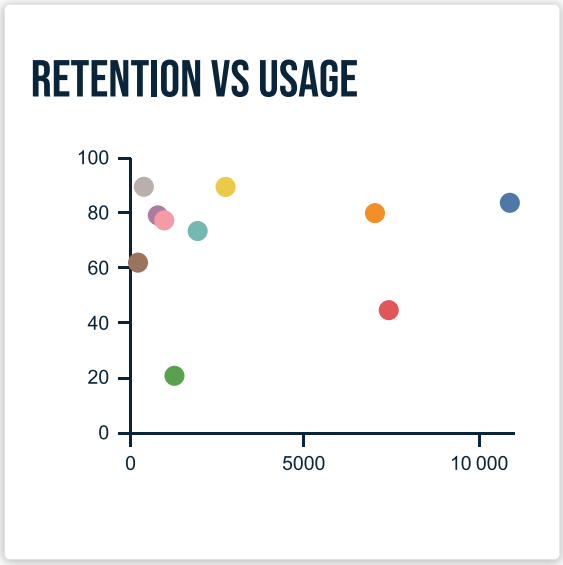


Рис. 8.6 Диаграмма рассеяния, построенная с применением подхода с передачей части дерева DOM под управление D3

Строить визуализации, используя подход с передачей части дерева DOM под управление библиотеки D3, относительно просто. С момента настройки атрибута `ref` мы можем использовать D3 так же, как в первой части этой книги. Этот подход выгоден, когда на разработку отводится мало времени или когда нужно быстро создать демонстрационную версию. Но важно помнить, что этот подход может отрицательно влиять на производительность проекта, поскольку библиотека React не может применить свои оптимизации к той части DOM, которую контролирует D3. В небольшом проекте, содержащем несколько визуализаций, эффект едва ли будет незаметен, но в крупных и сложных проектах производительность играет важную роль, и поэтому необходимо учитывать все нюансы, влияющие на нее.

В следующем разделе мы расскажем, как можно продолжать использовать D3, сохранив полный контроль над DOM за библиотекой React. Но прежде дадим несколько советов по реализации той же стратегии при использовании Angular и Svelte.

8.5.2 Angular

Чтобы применить ту же стратегию при работе с Angular, используем `ElementRef` – инструмент, с помощью которого Angular предоставляет прямой доступ к DOM. Для этого сначала импортируем `Ele-`

mentRef и ViewChild из основного модуля Angular, как показано на рис. 8.7. Затем в шаблоне HTML дадим прямую ссылку на элемент с помощью символа решетки (#) и имени по вашему выбору, например #myRef.

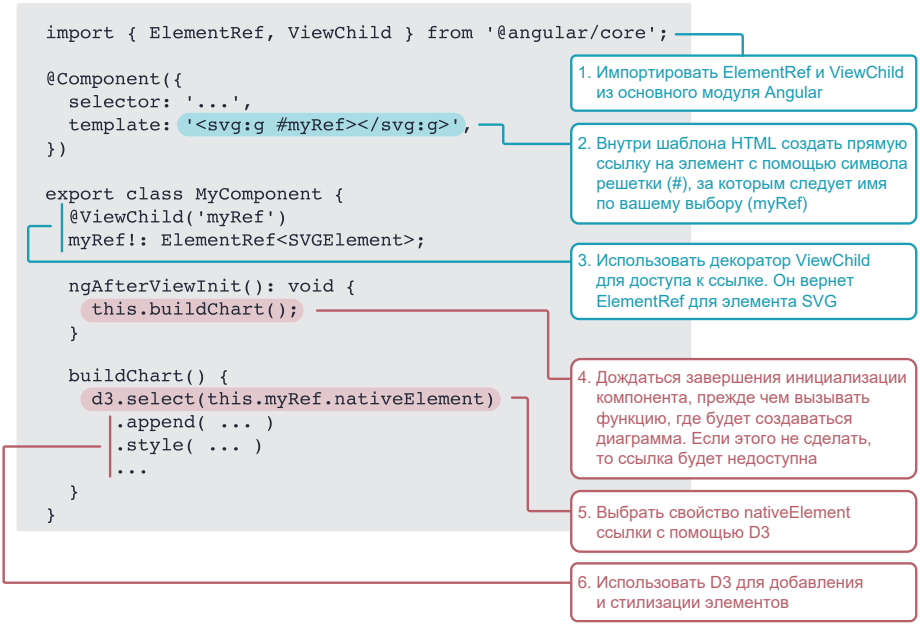


Рис. 8.7 Чтобы разрешить D3 управлять частью DOM в Angular, используется ElementRef. Мы передаем ссылку на элемент из шаблона HTML, используя символ решетки (#), за которым следует имя по нашему выбору. Затем внутри JavaScript-класса компонента передаем ссылку декоратору ViewChild. Наконец, после инициализации компонента выбираем свойство nativeElement ссылки с помощью D3 и используем эту библиотеку как обычно

Внутри JavaScript-класса компонента используем декоратор свойств ViewChild и передадим ему имя ссылки. Этот декоратор возвращает ElementRef, и мы указываем, что ссылка применяется к элементу SVG (<SVGElement>).

Далее нужно дождаться полной инициализации компонента, прежде чем пытаться манипулировать элементом, на который он ссылается. Для этого вызывается функция, в которой будем использовать D3, внутри обработчика событий жизненного цикла Angular ngAfterViewInit(). Наконец, с помощью D3 выбирается свойство nativeElement ссылки, и далее D3 используется как обычно. Чтобы увидеть, как эта стратегия применяется внутри реального приложения Angular, загляните в реализацию компонента scatterplot-d3-controlled в папке d3-with-angular в репозитории этой главы.

Эта стратегия позволяет использовать D3 привычным нам способом, но разработчики Angular не рекомендуют пользоваться ею. Ее следует использовать с большой осторожностью, потому что она мешает отображению DOM в Angular и может сделать приложение уязвимым для атак типа межсайтового скриптинга (cross-site scripting, XSS). В следующем разделе мы покажем более удачное решение.

Вам также следует знать о некоторых особенностях синтаксиса при работе с элементами SVG в Angular. Во-первых, при определении элементов SVG, отличных от контейнера SVG, требуется указывать префикс `svg:`, например `<svg:g></svg:g>`, `<svg:circle />` или `<svg:text></svg:text>`:

```
<svg>
  <svg:g>
    <svg:circle />
  </svg:g>
</svg>
```

Кроме того, имена атрибутов SVG должны снабжаться префиксом `attr`:

```
<svg:circle
  [attr.cx]="10"
  [attr.cy]="15"
  [attr.r]="5"
/>
```

Наконец, Angular использует TypeScript, поэтому по мере написания кода нужно объявлять типы. Вы можете установить определения типов для D3 с помощью NPM-пакета @types/d3.

8.5.3 Svelte

Svelte – последний писк моды в мире визуализации данных на момент написания этой книги. Легковесная архитектура этого фреймворка облегчает изучение и делает его эффективным инструментом для создания небольших, автономных проектов.

Подобно React и Angular, Svelte тоже позволяет с помощью ссылки передать часть дерева DOM под управление D3. Ссылка применяется к элементу DOM с помощью атрибута `bind:this`, которому присваивается имя ссылки. Затем, как только компонент будет смонтирован, можно выбрать ссылку с помощью D3 и начать использовать ее как обычно (см. рис. 8.8).

Чтобы увидеть применение этой стратегии в реальном приложении Svelte, загляните в реализацию компонента `scatterplotD3Controlled` в папке `d3-with-svelte` в репозитории этой главы.

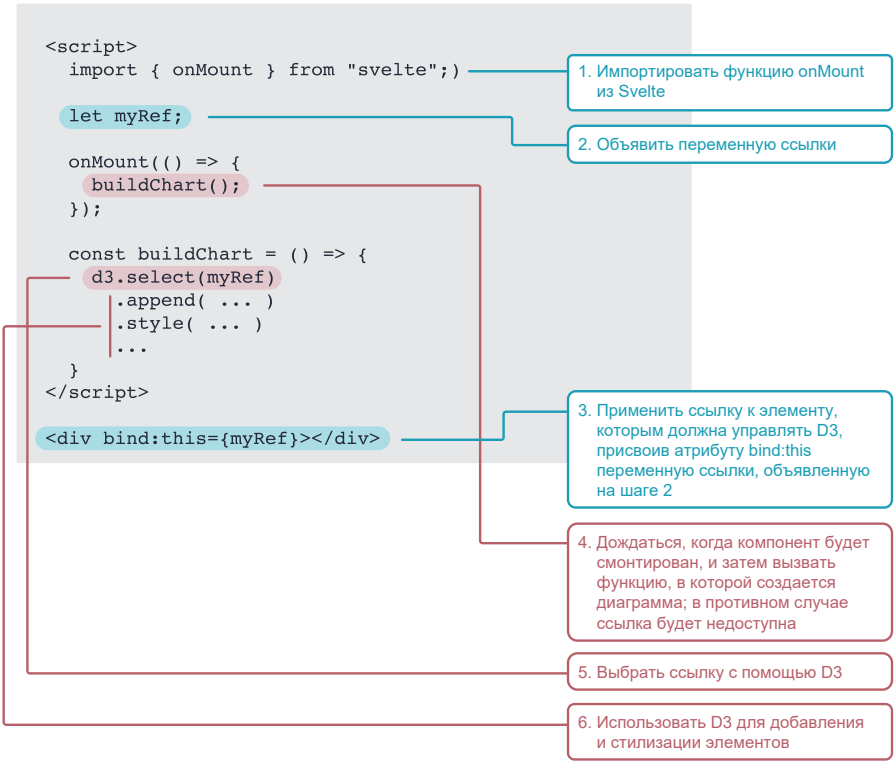


Рис. 8.8 Чтобы разрешить D3 управлять частью DOM в Svelte, мы объявляем ссылочную переменную в тегах <script> и с помощью bind:this ссылаемся на нее из элемента DOM, который будет управляться библиотекой D3. Затем вызовом `onMount()` ждем, пока компонент будет смонтирован, и вызываем функцию, конструирующую диаграмму. Внутри этой функции мы с помощью D3 выбираем ссылку и используем эту библиотеку как обычно

8.6 Использование D3 как вспомогательной библиотеки

В этом разделе мы обсудим более удачную стратегию использования D3 совместно с другими фреймворками, которая заключается в том, чтобы позволить фреймворку полностью контролировать DOM, а D3 использовать только как вспомогательную библиотеку. Эта стратегия подразумевает, что D3 не будет добавлять элементы в DOM, настраивать их атрибуты или определять их стилевые свойства. При применении этого подхода, например, мы не сможем использовать методы из модулей d3-selection, d3-axis или d3-transition.

Такой подход может показаться странным, и у кого-то из вас возникнет вопрос: зачем тогда вообще использовать D3? На самом деле мы все еще можем использовать множество мощных методов D3!

Например, выполнять все фоновые вычисления с помощью шкал или генераторов форм. А разметку и элементы SVG, составляющие визуализацию, можно генерировать средствами фреймворка.

Возьмите для примера диаграмму рассеяния, построенную в предыдущем разделе (рис. 8.9), и попробуйте мысленно перечислить элементы SVG, необходимые для ее рисования. В любом фреймворке вы сможете создать круг, нарисовать оси и создать метки осей с помощью текстовых элементов SVG. То есть мы почти ничего не теряем! В следующем подразделе мы заново построим диаграмму рассеяния, используя этот подход.

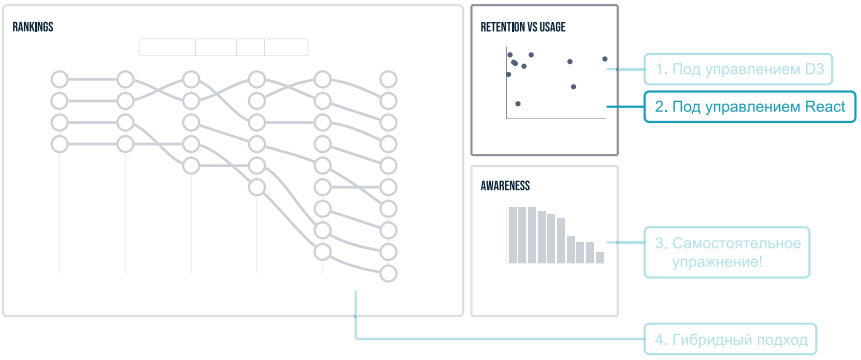


Рис. 8.9 Мы заново построим диаграмму рассеяния, отображающую зависимость лояльности от количества пользователей, управляя деревом DOM только с помощью React и используя D3 в качестве вспомогательной библиотеки

8.6.1 React

Чтобы построить диаграмму рассеяния, используя второй подход, откройте файл `Charts.js` и закомментируйте компонент `ScatterplotD3Controlled`. Импортируйте компонент `ScatterplotReactControlled` и отобразите его, передав те же свойства, что и раньше:

```
<ScatterplotReactControlled
  margin={margin}
  data={props.data.experience}
  colorScale={colorScale}
/>
```

Теперь откройте файл `ScatterplotReactControlled.js` и импортируйте компонент `ChartContainer`. Объявите константы `width` и `height` со значениями 300 и 245 соответственно. Рассчитайте соответствующие `innerWidth` и `innerHeight`, как показано в листинге 8.8. Наконец, добавьте компонент `ChartContainer` после заголовка `h2` и передайте ему необходимые свойства (`width`, `height` и `margin`).

Листинг 8.8 Подготовка ScatterplotReactControlled

```
import Card from '../UI/Card';
import ChartContainer from '../ChartComponents/
  ChartContainer';

const ScatterplotReactControlled = props => {
  const width = 300;
  const height = 245;
  const innerWidth = width - props.margin.left -
    props.margin.right;
  const innerHeight = height - props.margin.top -
    props.margin.bottom;

  return (
    <Card>
      <h2>Retention vs Usage</h2>
      <ChartContainer
        width={width}
        height={height}
        margin={props.margin}
      >
      </ChartContainer>
    </Card>
  )
};

export default ScatterplotReactControlled;
```

Импортировать компонент ChartContainer

Инициализировать константы width и height значениями 300 и 245 соответственно и вычислить innerWidth и innerHeight

Добавить компонент ChartContainer после заголовка h2 и передать ему необходимые свойства

Чтобы создать нашу диаграмму рассеяния, добавим в диаграмму элемент круга для каждого фреймворка в наборе данных. В предыдущем подходе мы бы использовали шаблон привязки данных, но поскольку теперь мы не позволяем D3 манипулировать деревом DOM, сделаем это вручную.

Круги являются основными элементами диаграммы и, скорее всего, будут использоваться в нескольких местах в проекте, поэтому создадим отдельный компонент, единственной задачей которого будет возврат элемента круга. Откройте файл `Circle.js` в папке `ChartComponents`. В настоящее время этот компонент возвращает элемент круга (`<circle />`), но не устанавливает его атрибуты. Было бы хорошо иметь возможность передать необходимые атрибуты через свойства. В листинге 8.9 мы присваиваем атрибутам `cx`, `cy`, `r` и `fill` круга значения входных свойств. Чтобы сделать компонент еще более универсальным, мы добавили возможность передачи значений для атрибутов `stroke` и `stroke-width`, но сделали их необязательными, применив тернарный оператор, который присваивает свойству `stroke` значение `"none"`, а свойству `stroke-width` значение `0`, если соответствующие значения не были переданы явно.

Листинг 8.9 Подготовка компонента Circle (Circle.js)

```
const Circle = props => {
  return (
    <circle
      cx={props.cx}
      cy={props.cy}
      r={props.r}
      fill={props.fill}
      stroke={props.stroke ? props.stroke : "none"}
      strokeWidth={props.strokeWidth ? props.strokeWidth : 0}
    />
  )
};

export default Circle;
```

Вернемся к компоненту `ScatterplotReactControlled` и импортируем компонент `Circle` и библиотеку `d3` в начале файла. Также объявляем горизонтальную и вертикальную линейные шкалы, которые будут отвечать за вычисление координат кругов на основе количества пользователей и процента лояльности.

Для создания кругов используем метод `map()`, добавляющий компонент `Circle` для каждого фреймворка в наборе данных. Чтобы помочь React следить за элементами DOM, передадим каждому компоненту `Circle` уникальный ключ. Их атрибуты `cx` вычисляются с помощью `xScale`, `cy` – с помощью `yScale` и `fill` – с помощью `colorScale`, после чего вычисленные значения передаются как входные свойства. Атрибут `r` имеет статическое значение `6`. Сохранив проект, вы должны увидеть круги на экране.

Листинг 8.10 Добавление кругов (ScatterplotReactControlled.js)

```
...
import * as d3 from 'd3';
import Circle from '../ChartComponents/Circle';

const ScatterplotReactControlled = props => {
  ...

  const xScale = d3.scaleLinear()
    .domain([0, d3.max(props.data, d => d.user_count)])
    .range([0, innerWidth])
    .nice();
  const yScale = d3.scaleLinear()
    .domain([0, 100])
    .range([innerHeight, 0]);
```

Импортировать библиотеку d3 и компонент Circle

Объявить две линейные шкалы для вычисления координат кругов на основе количества пользователей процента лояльности, который может варьироваться от 0 % до 100 %

```
  return (
    <Card>
      <h2>Retention vs Usage</h2>
      <ChartContainer ... >
        {props.data.map(framework => (
          <Circle
            key={`circle-${framework.id}`}
            cx={xScale(framework.user_count)}
            cy={yScale(framework.retention_percentage)}
            r={6}
            fill={props.colorScale(framework.id)}
          />
        ))}
      </ChartContainer>
    </Card>
  )
};
```

С помощью JavaScript-метода map() добавить компонент Circle для каждого фреймворка, присутствующего в наборе данных. Дать каждому уникальный ключ и настроить их атрибуты

```
export default ScatterplotReactControlled;
```

В завершение нужно создать оси без помощи генератора осей D3. У нас будет две оси: нижняя и левая. Откройте файл `Axis.js` в папке `ChartComponents`. Компонент `Axis`, объявленный в конце файла, получает входное свойство `type`, которое может иметь значение `"bottom"`, `"left"` или `"band"`. Если передано значение `"bottom"`, то вызывается функция `AxisBottom` и ей передаются остальные входные свойства, а если передано значение `"left"`, то вызывается функция `Axisleft`. Такая стратегия предполагает создание разных осей внутри одного файла, но точно так же мы могли бы реализовать их создание в отдельных файлах.

Давайте в `ScatterplotReactControlled` импортируем компонент `Axis` и дважды отобразим его внутри контейнера диаграммы, как показано в листинге 8.11. Первая ось имеет тип `"bottom"`, использует шкалу `xScale` и метку `"User Count"` (Количество пользователей). Также для правильного позиционирования осей нужно передать `innerWidth` и `innerHeight`. Вторая ось имеет тип `"left"`, использует шкалу `yScale` и метку `"Retention %"` (Лояльность %). Чуть ниже вы увидите, как эти свойства будут использоваться для рисования осей.

Листинг 8.11 Вызов Axis (ScatterplotReactControlled.js)

```
...
import Axis from '../ChartComponents/Axis';

const ScatterplotReactControlled = props => {
  ...

  return (
    <Card>
```

Импортировать компонент Axis

```
      <h2>Retention vs Usage</h2>
      <ChartContainer ... >
        <Axis
          type="bottom"
          scale={xScale}
          innerWidth={innerWidth}
          innerHeight={innerHeight}
          label={"User Count"}
        />
        <Axis
          type="left"
          scale={yScale}
          innerWidth={innerWidth}
          innerHeight={innerHeight}
          label={"Retention %"}
        />
        {props.data.map(framework => (
          <Circle
            ...
          />
        ))}
      </ChartContainer>
    </Card>
  )
};

export default ScatterplotReactControlled;
```

Отобразить компонент Axis дважды, первый раз, чтобы нарисовать нижнюю ось, а второй – левую. Передать соответствующую шкалу, метку, innerWidth и innerHeight как входные свойства

В главе 4 мы говорили, что при создании оси D3 генерирует три типа элементов: линию (элемент `path`), текстовый элемент и короткую линию для каждой риски с меткой, которые отображаются вместе с осью. На рис. 8.10 показана нижняя ось из диаграммы рассеяния, созданной в разделе 8.5. Эту ось мы легко можем нарисовать вручную.

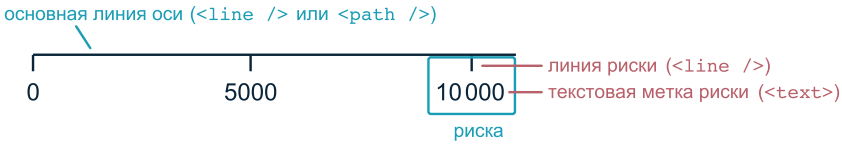


Рис. 8.10 Оси, созданные библиотекой D3, состоят из основной линии, меток и коротких линий рисок

Вернемся к файлу `Axis.js` и начнем с создания нижней оси внутри функции `AxisBottom`. Сначала определим, сколько рисок на оси нужно отобразить. В листинге 8.12 мы указываем, что риски будут отображаться через каждые 100 пикселей. Если разделить значение `innerWidth`, которое равно 300 пикселям, на 100, то получится три риски. Сохраним это число в константе `numberOfTicks`.

В D3 непрерывные шкалы имеют метод `ticks()`, принимающий число и возвращающий массив с рисками и их значением из предметной области. Там же в листинге 8.12 мы вызываем этот метод как `props.scale.ticks(numberOfTicks)` и получаем массив `[0, 5000, 10000]`, который сохраняем в константе `ticks`.

Чтобы нарисовать ось, смещаем в нижнюю часть диаграммы элемент группы, который уже существует в операторе `return`, и присваиваем ему имя класса `"axis"`. В файле `Axis.css` уже определено несколько стилей для этого класса. Затем добавляем элемент линии, простирающийся слева направо, от начала до конца.

Для отображения рисок в массиве `ticks` мы добавляем группу, короткую вертикальную линию и текстовый элемент. Наконец, если передается входное свойство с меткой оси, то мы показываем ее под осью.

Листинг 8.12 Создание нижней оси (Axis.js)

Вычислить количество рисок для отображения. Создать массив, содержащий метку для каждой риски, вызвав метод tick(), который можно применить к любой непрерывной шкале D3

```
const AxisBottom = props => {
  const numberOfTicks = props.innerWidth / 100;
  const ticks = props.scale.ticks(numberOfTicks);

  return (
    <g className="axis" transform={`translate(0,
      ${props.innerHeight})`} >
      <line x1={0} y1={0} x2={props.innerWidth}
        y2={0} />
      {ticks.map(tick => (
        <g key={tick} transform={`translate(
          ${props.scale(tick)}, 0)`}>
          <line x1={0} y1={0} x2={0} y2={5} />
          <text x={0} y={20} textAnchor="middle" >
            {tick}
          </text>
        </g>
      ))}
      {props.label &&
        <text
          className="axis-label"
          textAnchor="middle"
          transform={`translate(${props.innerWidth /
            2}, 45)`}
        >
          {props.label}
        </text>
      }
    </g>
  );
};
```

Переместить элемент группы в нижнюю часть диаграммы и присвоить ему имя класса «axis»

Добавить горизонтальную линию, простирающуюся от левого края диаграммы до правого

Для каждой риски в массиве ticks добавить короткую вертикальную линию и текстовый элемент

Если компоненту передано входное свойство label, то отобразить метку с текстом из этого свойства под осью

В листинге 8.13 аналогичным образом создается левая ось. Главное отличие: риски отображаются через каждые 50 пикселей и основная метка поворачивается на 90°.

Листинг 8.13 Создание левой оси (Axis.js)

```
const AxisLeft = props => {
  const numberOfTicks = props.innerHeight / 50;
  const ticks = props.scale.ticks(numberOfTicks);

  return (
    <g className="axis">
      <line x1={0} y1={props.innerHeight} x2={0}
        y2={0} />
      {ticks.map(tick => (
        <g key={tick} transform={`translate(0,
          ${props.scale(tick)})`}>
          <line x1={-5} y1={0} x2={0} y2={0} />
          <text x={-10} y={0} textAnchor="end"
          ➥ alignmentBaseline="middle">
            {tick}
          </text>
        </g>
      ))}
      {props.label &&
        <text
          textAnchor="middle"
          transform={`translate(-42,
            ${props.innerHeight / 2})
            ➥ rotate(-90)`}
        >
          {props.label}
        </text>
      }
    </g>
  );
};
```

Вычислить количество рисок для отображения. Создать массив, содержащий метку для каждой риски, вызвав метод tick(), который можно применить к любой непрерывной шкале D3

Добавить вертикальную линию, которая простирается от низа до верха диаграммы

Для каждой риски в массиве ticks добавить короткую вертикальную линию и текстовый элемент

Если компоненту передано входное свойство label, то отобразить метку слева от диаграммы, повернув ее на 90 градусов

По завершении диаграмма рассеяния должна выглядеть, как показано на рис. 8.11.

8.6.2 Angular и Svelte

Стратегия использования D3 исключительно в роли вспомогательной библиотеки не зависит от фреймворка, управляющего деревом DOM, будь то React, Angular или Svelte. Освоив фреймворк и его синтаксис,

вы сможете построить любую диаграмму, как мы делали выше в этом разделе. Примеры построения диаграммы рассеяния с применением второй стратегии в Angular и Svelte вы найдете в компонентах `scatterplot-angular-controlled` и `ScatterplotSvelteControlled` в папках `d3-with-angular` и `d3-with-svelte` в репозитории этой главы.

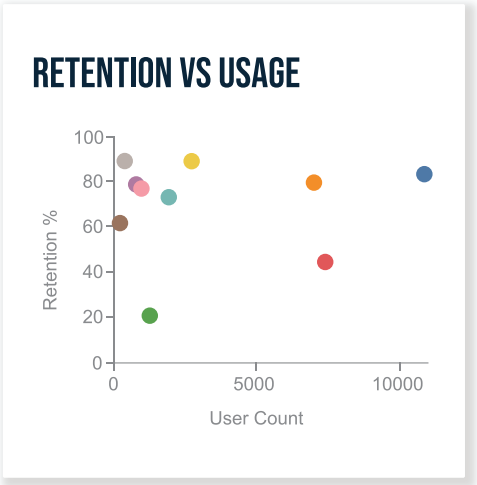
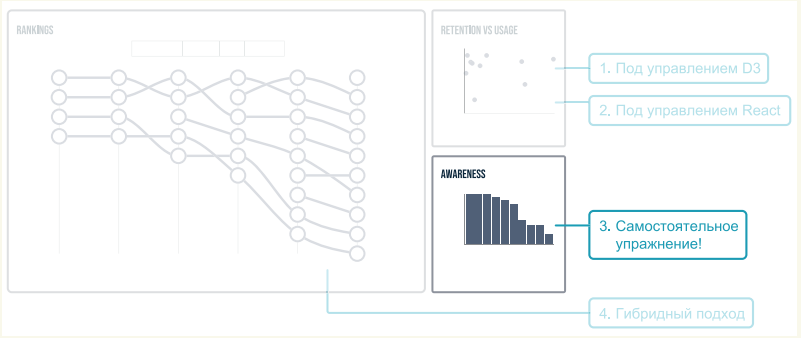


Рис. 8.11 Диаграмма рассеяния, построенная с использованием D3 исключительно в роли вспомогательной библиотеки

Упражнение: создание столбчатой диаграммы

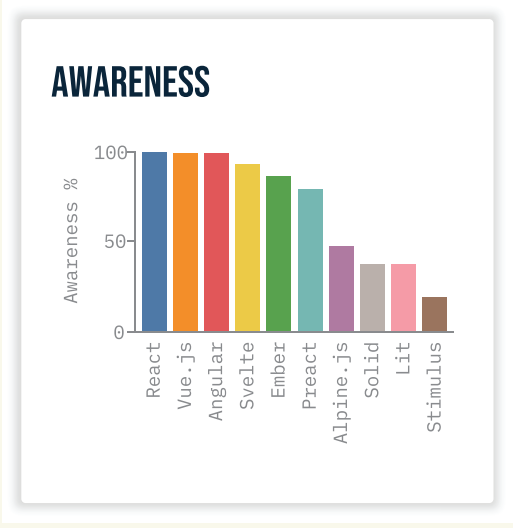
Давайте закрепим полученные знания, построив столбчатую диаграмму осведомленности разработчиков о каждом фреймворке, как показано на рисунке ниже.



Теперь ваша очередь попробовать свои силы! Создание элементов диаграммы и осей вручную может показаться непростым делом, однако это упражнение поможет вам прояснить процесс

Вот несколько советов.

1. Создайте компонент, рисующий прямоугольник (`Rectangle.js`), и используйте его для создания каждого прямоугольника в столбчатой диаграмме (в `BarChart.js`).
2. Установите ширину и высоту столбчатой диаграммы равными 300 и 245 пикселей соответственно. Увеличьте нижний отступ, чтобы разместить метки (мы использовали отступ высотой 85 пикселей).
3. Используйте интервальную шкалу для расчета горизонтального местоположения каждого столбца.
4. Чтобы улучшить читаемость визуализации, отсортируйте столбцы в порядке убывания с помощью JavaScript-метода `sort()`.
5. При создании нижней оси передайте свойство `type` со значением `"band"` и сгенерируйте ось в функции `AxisBandBottom()`, доступной в `Axis.js`.
6. Интервальная шкала не является непрерывной, поэтому у нее нет метода `ticks()`. Чтобы сгенерировать метку для каждого фреймворка, передайте массив их имен компоненту `Axis`.
7. Метки на нижней оси вы должны будете повернуть. Для этого можно использовать атрибут `transform`, чтобы указать смещение и угол поворота.



Законченная столбчатая диаграмма осведомленности разработчиков о каждом фреймворке

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в листинге D.12 и в папке `8.6.b-Bar_chart/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

8.6.3 Создание кривых

До сих пор мы использовали в наших визуализациях простые геометрические фигуры, такие как круги, прямоугольники и линии. А как быть, если понадобится создать кривую с использованием генератора фигур D3? Поскольку единственная задача генератора фигур – возвращать значение для атрибута `d` элемента `path`, мы можем продолжать использовать его и при применении второй стратегии. Продемонстрируем эту возможность, построив диаграмму рейтингов, которую можно увидеть на рис. 8.1 и в онлайн-версии проекта (http://mng.bz/yZlB). Эта диаграмма показывает рейтинг каждого фреймворка структуры с точки зрения удовлетворенности пользователей. Фреймворк, занимающий первое место, расположен вверху, а занимающий последнее место – внизу. Для начала в файле `Charts.js` передадим данные и цветовую шкалу компоненту `Rankings`:

```
<Rankings
  margin={margin}
  data={props.data}
  colorScale={colorScale}
/>
```

Затем перейдем к компоненту `Curve` и объявим генератор линий, как показано в листинге 8.14. Предполагается, что генератор будет получать `xScale` и `yScale` в качестве свойств, а также ключи доступа для использования в функциях `x()` и `y()`, которыми в нашем случае являются `"year"` и `"rank"`.

Поскольку данные за первые годы имеются не для всех фреймворков, используем функцию доступа `defined()`, которая будет обрабатывать годы без данных. Мы также выбираем функцию интерполяции `d3.curveMonotoneX`.

При добавлении элемента `path` мы определяем значение его атрибута `d`, вызывая генератор `lineGenerator(props.data)`. Атрибуты `stroke` и `strokeWidth` передаются как входные свойства.

Листинг 8.14 Генератор кривых (Curve.js)

```
import * as d3 from "d3";

const Curve = props => {
  const lineGenerator = d3.line()
    .x(d => props.xScale(d[props.xAccessor]))
    .y(d => props.yScale(d[props.yAccessor]))
    .defined(d => d[props.yAccessor] !== null)
    .curve(d3.curveMonotoneX);

  return (
```

Объявить функцию генератора линии. Использовать шкалы и ключи, переданные как свойства, для настройки функций доступа `x()` и `y()`. Предусмотреть обработку отсутствующих данных с помощью метода defined()

```
    <path
      d={lineGenerator(props.data)}
      fill="none"
      stroke={props.stroke}
      strokeWidth={props.strokeWidth}
    />
  );
};

export default Curve;
```

Добавить элемент path и установить его атрибут d, вызвав функцию генератора линии и передав ей входное свойство data

Компонент `Curve` вызывается в файле `Rankings.js`. Обратите внимание, что в этом файле имеется массив с именем `rankingFilters`, содержащий информацию для каждой кнопки, отображаемой над диаграммой. В компоненте также есть переменная состояния с именем `activeFilter`, которая инициализируется значением `satisfaction`. Эти фильтры мы реализуем в следующем разделе и обновим визуализацию рейтингов соответствующим образом.

А пока объявим наши шкалы. Если обратиться к рис. 8.1 или онлайн-версии проекта (http://mng.bz/yZlB), то можно заметить, что годы с 2016 по 2021 распределены горизонтально, а фреймворки распределены вертикально по рангу. В обоих случаях мы можем использовать точечную шкалу, как показано в листинге 8.15.

Для каждого фреймворка мы вызываем компонент `Curve` и передаем ему требуемые свойства. Обратите внимание, что данные, передаваемые компоненту `Curve`, определяются с помощью переменной состояния `activeFilter`, что означает, что в настоящее время передается массив, доступный под ключом `satisfaction`, представляющий ранг текущего фреймворка с точки зрения удовлетворенности пользователей в период с 2016 по 2021 год.

Листинг 8.15 Отображение кривых рангов (Rankings.js)

```
import * as d3 from 'd3';
import Curve from '../ChartComponents/Curve';

const Rankings = props => {
  ...

  const xScale = d3.scalePoint()
    .domain(props.data.years)
    .range([0, innerWidth]);
  const yScale = d3.scalePoint()
    .domain(d3.range(1, props.data.ids.length + 1))
    .range([0, innerHeight]);

  return (
    <Card>
      <h2>Rankings</h2>
```

Объявить две точечные шкалы: одну, распределяющую годы по горизонтали, и другую, размещающую ранги по вертикали

```
      <RankingFilters ... />
      <ChartContainer ... >
        {props.data.experience.map((framework, i) => (
          <g key={`curve-${framework.id}`}>
            <Curve
              data={framework[activeFilter]}
              xScale={xScale}
              yScale={yScale}
              xAccessor="year"
              yAccessor="rank"
              stroke={props.colorScale(framework.id)}
              strokeWidth={5}
            />
          </g>
        ))}
      </ChartContainer>
    </Card>
  )
};
```

Вызвать компонент Curve для каждого фреймворка и передать данные, связанные с activeFilter, а также другие необходимые свойства. Обратите внимание, что компонент Curve заключен в элемент группы. Чуть ниже мы добавим метки в эти группы

Чтобы прояснить, какая кривая какой фреймворк представляет, добавим метки слева и справа от диаграммы. В листинге 8.16 показана реализация компонента `Label`, который задает атрибуты элемента `text` на основе своих входных свойств. Поскольку метки будут располагаться по обе стороны диаграммы, нужно также установить свойство `textAnchor`. Для улучшения читаемости мы присваиваем свойству `fontWeight` значение `"bold"`.

Листинг 8.16 Компонент Label (Label.js)

```
const Label = props => {
  return (
    <text
      x={props.x}
      y={props.y}
      fill={props.color}
      textAnchor={props.textAnchor}
      alignmentBaseline="middle"
      style={{ fontWeight: "bold" }}
    >
      {props.label}
    </text>
  );
};

export default Label;
```

Код в листинге 8.17 вызывает компонент `Label` дважды, если фреймворк имеет данные за первый год (2016), и один раз в про-

тивном случае. Первая метка с названием фреймворка отображается слева от диаграммы с `textAnchor "end"`, а вторая – справа с `textAnchor "start"`.

Листинг 8.17 Добавление меток слева и справа от диаграммы (Rankings.js)

```
...
import Label from '../ChartComponents/Label';

const Rankings = props => {
  ...

  return (
    <Card>
      <h2>Rankings</h2>
      <RankingFilters ... />
      <ChartContainer ... >
        {props.data.experience.map((framework, i) => (
          <g key={`curve-${framework.id}`}>
            <Curve ... />
            {framework[activeFilter][0].rank &&
              <Label
                x={-25}
                y={yScale(framework[activeFilter][0]
                  .rank)}
                color={props.colorScale(framework.id)}
                label={framework.name}
                textAnchor={"end"}
              />
            }
            <Label
              x={innerWidth + 25}
              y={yScale(framework[activeFilter]
              ➥ [framework[activeFilter].length -
                    1].rank)}
              color={props.colorScale(framework.id)}
              label={framework.name}
              textAnchor={"start"}
            />
          </g>
        ))}
      </ChartContainer>
    </Card>
  )
};
```

Если для фреймворка есть данные за первый год (2016), то отобразить метку слева от диаграммы с атрибутом textAnchor "end"

Отобразить метку для каждого фреймворка справа от диаграммы

Наконец, вслед за кривыми добавим для каждого года пунктирную вертикальную линию и текстовый элемент, как показано в листинге 8.18. Текстовые элементы с годами добавляются явно, пото-

му что в этом примере так проще, но вообще для этого можно было бы настроить компонент `Axis`. Оба элемента заключаются в группу с классом `"axis"`, чтобы применить к ним стили, объявленные в `Axis.css`.

Листинг 8.18 Добавление сетки и меток (Rankings.js)

```
const Rankings = props => {
  ...

  return (
    <Card>
      <h2>Rankings</h2>
      <RankingFilters ... />
      <ChartContainer ... >
        {props.data.years.map(year => (
          <g
            key={`line-year-${year}`}
            className="axis"
            transform={`translate(${xScale(year)}, 0)`}
          >
            <line
              x1={0}
              y1={innerHeight}
              x2={0}
              y2={0}
              strokeDasharray={"6 4"}
            />
            <text
              x={0}
              y={innerHeight + 30}
              textAnchor="middle"
            >
              {year}
            </text>
          </g>
        ))}
        {props.data.experience.map((framework, i) => (
          <g key={`curve-${framework.id}`}>
            <Curve ... />
            ...
          </g>
        ))}
      </ChartContainer>
    </Card>
  )
};
```

Для каждого года в наборе данных добавить элемент группы. Присвоить ему класс "axis", чтобы повторно использовать стили, уже объявленные в Axis.css, и переместить каждую группу в местоположение, соответствующее году

Для каждого года добавить пунктирную вертикальную линию

Для каждого года добавить метку под диаграммой

По завершении диаграмма рейтингов должна выглядеть, как показано на рис. 8.12.

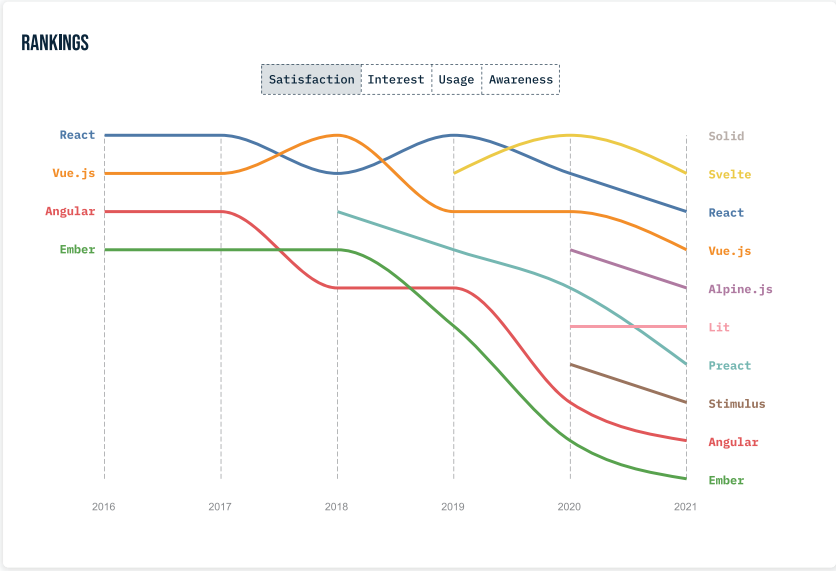
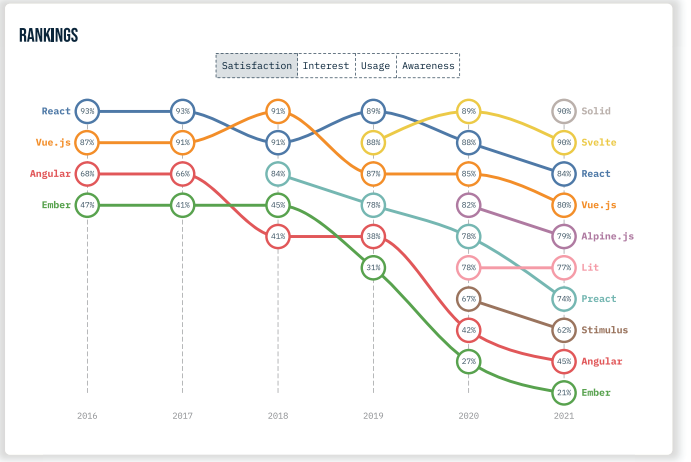


Рис. 8.12 Диаграмма рейтингов с названиями фреймворков

Упражнение: добавление значков в визуализацию рейтингов

Для завершения статической версии нашей информационной панели осталось добавить всего один элемент: значки, сообщающие процент разработчиков, ответивших в том или ином году, что они удовлетворены фреймворком. Результат показан на рисунке ниже. Попробуйте реализовать эти значки самостоятельно, выполнив следующие инструкции.

1. Каждый значок состоит из трех элементов:
   - а элемент группы, который перемещается в нужную позицию с помощью CSS-свойства `transform`;
   - б круг с белой заливкой и толщиной границы в 3 пикселя, цвет границы соответствует связанному фреймворку;
   - в текстовый элемент, отображающий процент для выбранного фильтра. В настоящее время выбран фильтр `Satisfaction`, но в следующем разделе он будет меняться. Обязательно примите это во внимание при передаче текста в компонент значка.
2. Мы предлагаем создавать значки в компоненте `Badge`, доступном в папке `/UI`.
3. Для отображения кругов используйте компонент `Circle`, который уже использовался для рисования диаграммы рассеяния. Круги должны иметь радиус 18 пикселей.
4. Нелишним будет округлить проценты до целых чисел с помощью функции `Math.round()`.
5. Для текста мы в своем проекте использовали цвет `#374f5e` и жирный шрифт с размером 12 пикселей.



Законченная диаграмма рейтингов со значками

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в листинге D.13 и в папке `8.6.d-Rankings_with_badges/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

8.7 Гибридный подход

Чтобы завершить нашу информационную панель, добавим обработку событий нажатий кнопок, находящихся в верхней части диаграммы рейтингов. В ходе реализации решения мы обсудим еще одну, последнюю стратегию использования D3 с фреймворком: гибридную, упомянутую на рис. 8.13.

Но сначала откройте файл `RankingFilters.js` и добавьте событие `onClick` в компонент `Button`, как показано в следующем фрагменте. В качестве обработчика события укажите значение входного свойства `onFilterSelection`, полученное от родительского компонента. Обратите внимание, что компонент `Button` уже возвращает свой `id` при щелчке (см. `Button.js`):

```
<Button
  key={filter.id}
  id={filter.id}
  label={filter.label}
  isActive={props.activeFilter === filter.id ? true : false}
  onClick={props.onFilterSelection}
/>
```

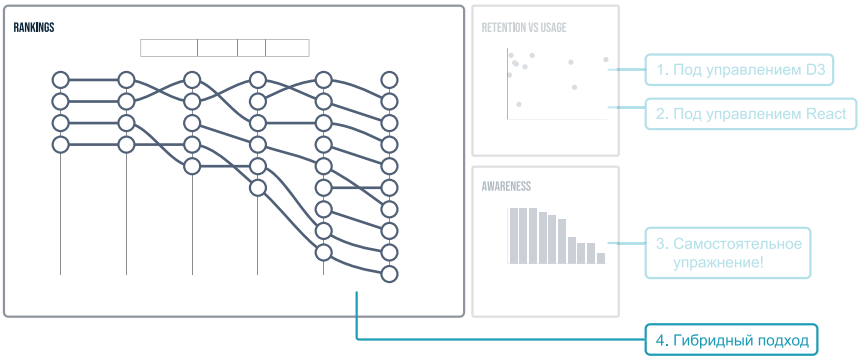


Рис. 8.13 При создании диаграммы рейтингов мы используем гибридный подход: сама диаграмма целиком построена с помощью React, но чтобы обеспечить плавность переходов, значение для атрибута d вычисляется с помощью D3

Затем перейдите к родительскому компоненту в `Rankings.js` и добавьте входное свойство `onFilterSelection` в компонент `RankingFilters`. Это свойство должно ссылаться на функцию с именем `filterSelectionHandler`:

```
<RankingFilters
  filters={rankingFilters}
  activeFilter={activeFilter}
  onFilterSelection={filterSelectionHandler}
/>
```

Там же, в компоненте `Rankings`, объявите функцию `filterSelectionHandler`, которая принимает идентификатор выбранного фильтра. Если переменная состояния `activeFilter` не равна полученному идентификатору, то функция должна обновить состояние, сохранив полученный идентификатор, используя для этого `setActiveFilter()`:

```
const filterSelectionHandler = (id) => {
  if (activeFilter !== id) {
    setActiveFilter(id);
  }
};
```

Закончив, сохраните проект, откройте его в браузере и попробуйте пощелкать на кнопках. Диаграмма рейтингов должна обновляться соответствующим образом.

Итак, фильтрация работает, но нам не хватает плавной смены кривых и положений их меток и значков. Поскольку D3 упрощает реализацию анимационных эффектов, мы могли бы подумать, что имеет смысл разрешить D3 управлять этими линиями и применить метод `transition()` к их атрибутам `d`. Поскольку это небольшой проект, такое решение не должно существенно повлиять на его произво-

дительность. Но имейте в виду, что в общем случае решение должно зависеть от конкретного проекта, команды и приоритетов.

В листинге 8.19 мы импортируем ловушки `useRef` и `useEffect` из React. Затем инициализируем ловушку `useRef` и сохраняем ее в константе `pathRef`. Добавляем атрибут `ref` в элемент `path` и присваиваем ему эту константу. Обратите внимание, что мы также удалили атрибут `d` из элемента `path`.

Наконец, в ловушке `useEffect` мы передаем анонимную функцию, которая с помощью D3 выбирает текущий экземпляр `pathRef`, применяет функцию перехода и обновляет атрибут `d`, вызывая генератор линии. Поскольку эта анонимная функция зависит от данных, полученных во входных свойствах, и генератора линии, мы должны передать их как зависимости. Последние две не изменяются, но каждый раз, когда данные во входных свойствах, передаваемых в компонент `Curve`, изменяются, будет вызываться анонимная функция в `useEffect`.

Листинг 8.19 Реализация плавной смены кривых (Curve.js)

```
import { useRef, useEffect } from "react";
import * as d3 from "d3";

const Curve = props => {
  const lineGenerator = d3.line()
    .x(d => props.xScale(d[props.xAccessor]))
    .y(d => props.yScale(d[props.yAccessor]))
    .defined(d => d[props.yAccessor] !== null)
    .curve(d3.curveMonotoneX);

  const pathRef = useRef();
  useEffect(() => {
    const path = pathRef.current;
    d3.select(path)
      .transition()
      .duration(400)
      .ease(d3.easeCubicOut)
        .attr("d", lineGenerator(props.data));
  }, [props.data, lineGenerator]);

  return (
    <path
      ref={pathRef}
      fill="none"
      stroke={props.stroke}
      strokeWidth={props.strokeWidth}
    />
  );
};

export default Curve;
```

Импортировать ловушки useRef и useEffect из React

Объявить ловушку useRef и сохранить ее в константе с именем pathRef

Создать ловушку useEffect и передать ей анонимную функцию, которая выбирает текущий экземпляр pathRef. Применить переход к элементу path и обновить его атрибут d, вызвав генератор строк

Поскольку анонимная функция зависит от данных во входных свойствах, функции перехода и генератора линий должны передаваться как зависимости. Последние две не меняются, но при каждом обновлении данных будет запускаться анонимная функция внутри useEffect()

Применить к элементу path атрибут ref со ссылкой на pathRef. Удалить атрибут d

Плавный переход также следует применить к значкам и меткам с названиями фреймворков. Мы могли бы использовать механизм переходов в D3, но давайте подумаем, нет ли более простого решения. Например, метки можно переместить с помощью CSS-свойства `transform` и применить CSS-переход. Это не только проще, но и, вероятно, лучше с точки зрения производительности, учитывая, что меток и значков несколько.

В следующем фрагменте мы расширили текстовый элемент, возвращаемый компонентом `Label`: установили его атрибуты `x` и `y` в значение `0`, заключили в элемент группы и позиционировали группу с помощью CSS-свойства `transform`. Мы также дали группе имя класса `"label"`:

```
import "./Label.css";

<g
  className="label"
  style={{ transform: `translate(${props.x}px, ${props.y}px)` }}
>
  <text
    x={0}
    y={0}
    fill={props.color}
    textAnchor={props.textAnchor}
    alignmentBaseline="middle"
    style={{ fontWeight: "bold" }}
  >
    {props.label}
  </text>
</g>
```

> **ПРИМЕЧАНИЕ** CSS-свойство `transform` применяется к группе, а не к текстовому элементу, из-за ошибки в браузере Safari, вследствие которой это свойство применяется к текстовому элементу неправильно. В других браузерах это свойство с тем же успехом можно применить и к текстовому элементу.

Затем создаем файл с именем `Label.css` и импортируем его в компонент. В этом файле мы устанавливаем свойство `transition` меток следующим образом:

```
.label {
  transition: transform 400ms cubic-bezier(0.33, 1, 0.68, 1);
}
```

Наконец, поскольку элемент группы, обертывающий значки, уже позиционирован, нам остается только присвоить ему имя класса `"label"`, как в предыдущем фрагменте. Благодаря тому что стили

в React по умолчанию являются глобальными, переход будет применен к элементу автоматически.

Теперь, получив представление о трех подходах к интеграции D3 с другими фреймворками пользовательских интерфейсов, вы все еще можете пребывать в сомнениях, какой из них выбрать. Правильный ответ во многом зависит от типа проекта и ваших приоритетов, однако возьмем на себя смелость дать несколько рекомендаций.

1. Первый подход, в котором мы позволяем библиотеке D3 управлять частью дерева DOM, в основном предназначен для быстрой разработки прототипов и проектов с небольшим числом визуализаций, где производительность не является главной проблемой.
2. Второй подход, когда деревом DOM манипулирует только фреймворк пользовательского интерфейса, а D3 используется как вспомогательная библиотека, предпочтительнее в большинстве случаев. Он не только повышает производительность в крупных проектах, но и упрощает обслуживание в больших командах. В таких командах большинство разработчиков не знают D3. Им будет проще понять код, в котором элементы SVG создаются вручную, а не с помощью шаблона привязки данных или генераторов осей.
3. Гибридный подход, в котором библиотеке D3 позволяется манипулировать частью дерева DOM, следует использовать с осторожностью и только если он дает явное преимущество. Если вы планируете использовать D3 для выполнения плавных переходов, то подумайте – возможно, для этой цели лучше использовать переходы или анимацию CSS либо даже библиотеку анимационных эффектов, созданную специально для вашего фреймворка.

На этом мы заканчиваем работу над информационной панелью! В следующей главе мы сделаем ее адаптивной.

Интервью с Коннором Ротшильдом

Коннор Ротшильд (Connor Rothschild) – инженер по визуализации данных.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

В колледже я специализировался на политологии и только на третьем курсе решил попробовать свои силы в визуализации данных. Вдохновленный такими публикациями, как *The Pudding* и *New York Times*, я захо-

тел узнать, как создавать захватывающие визуальные истории. Я начал создавать статические диаграммы с помощью R и ggplot2 и добавлять их в отчеты для аналитического центра, расположенного в моем университете.

Постепенно я вошел в интерактивное пространство и начал изучать инструменты веба (HTML, CSS, JavaScript). Впервые я познакомился с D3, когда прочитал книги *Fullstack D3 and Data Visualization* Амелии Ваттенбергер (Amelia Wattenberger) и *Interactive Data Visualization for the Web* Скотта Мюррея (Scott Murray).

После (довольно мучительного) самостоятельного изучения D3 я попробовал свои силы в создании нескольких историй на основе данных из сети, например «How Much Does Kid Cudi Hum?» [https://connorrothschild.github.io/cudi-hums/] и «The Bob Ross Virtual Art Gallery» [https://connorrothschild.github.io/bob-ross-art-gallery/]. Эти проекты в конечном итоге привлекли внимание других людей, первый выиграл Pudding Cup в 2020 году, а второй принес мне запрет от Bob Ross, Inc.! (Серьезно… спросите меня как-нибудь за чашкой кофе.)

Окончив колледж, я некоторое время работал визуальным журналистом в Axios, а в 2021 году стал соучредителем Moksha Data Studio [https://mokshadata.studio/]. Мы работаем с некоммерческими организациями, редакциями и социально ориентированными компаниями, которые желают рассказывать визуально убедительные истории в интернете.

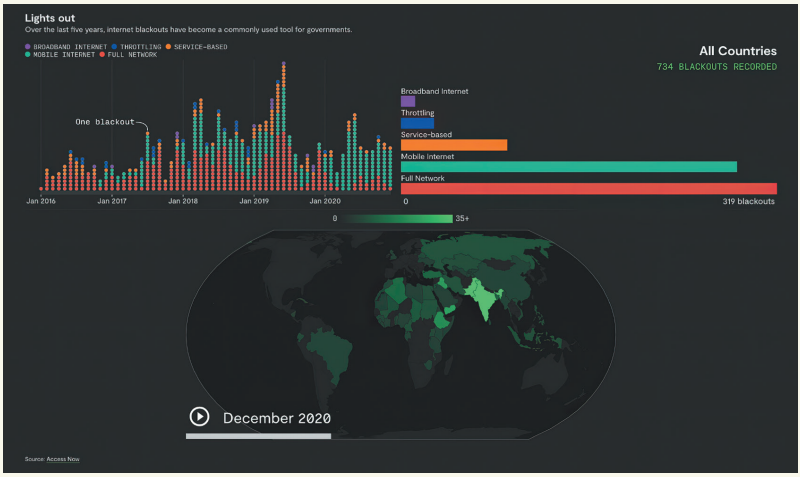
Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашем рабочем процессе визуализации данных?

О да! Я страстный сторонник использования D3 в сочетании с другими фреймворками JavaScript. В частности, я предпочитаю Svelte. Библиотеку D3 я использую в основном для математических вычислений, манипулирования массивами и масштабирования данных, а Svelte – для отображения элементов на экране.

Вы создали курс в содружестве с Newline под названием «Better Data Visualization with Svelte», и он мне очень понравился. Расскажите, чем вам понравилось сочетание этих двух технологий.

Как разработчику-самоучке синтаксис работы с выборками в D3 (`.enter`, `.exit`, `.join` и т. д.) кажется мне немного пугающим. Другие элементы D3, такие как шкалы D3 и функции для работы с массивами, выглядят намного проще и понятнее. Svelte избавляет от необходимости вручную манипулировать объектной моделью документа (DOM) и позволяет писать разметку непосредственно (используя встроенные инструменты, такие как блоки {#each} и встроенная привязка данных). Я считаю, что мы можем получить лучшее из обоих миров, используя D3 для математики и манипуляции данными, а Svelte – для работы с DOM.

Представленный проект: «In the Dark» (https://restofworld.org/2022/blackouts/)



Проект исследует количество отключений интернета в период с 2016 по 2020 год по типу и стране

Опишите, пожалуйста, ваш проект «In the Dark», который вы создали для сайта «Rest of World» (https://restofworld.org).

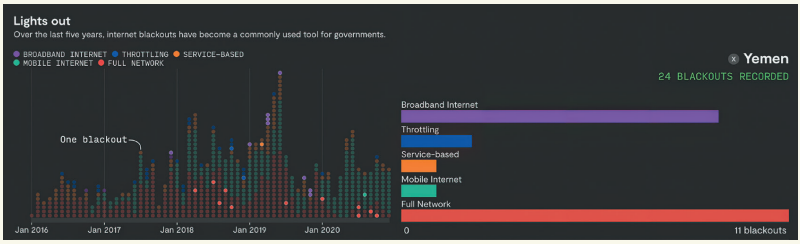
Это был по-настоящему увлекательный проект, над которым мы работали вместе с Rest of World. Он рассказывает историю отключений электроэнергии по всему миру. Нам дали различные наборы данных, охватывающие типы и частоту отключений электроэнергии по странам с течением времени.

Расскажите, как создавался этот проект.

Мы тесно сотрудничали с автором истории и дизайнером «Rest of World», чтобы создать серию тематически согласованных визуализаций данных. Мы встречались еженедельно, чтобы показать прогресс в разработке каждой диаграммы и получить новые замечания и пожелания. Данные уже имелись у нас, и мы имели полную творческую свободу в выборе дизайна и типов диаграмм.

Вы использовали точечный график для визуализации каждого события отключения электроэнергии по отдельности. А для расчета вертикального положения каждой точки вы создали свою функцию или использовали силовую компоновку D3 с жестким ограничением для горизонтального положения?

Для вычисления позиций каждой точки по осям x и y мы создали свои линейные и точечные шкалы! Мы рассматривали возможность применения силовой компоновки, но, учитывая статичный характер диаграммы, мы выбрали этот более жесткий подход к позиционированию.



Благодаря перекрестному выделению мы можем исследовать события, происшедшие в определенной стране

В этом проекте карта действует как фильтр, позволяя исследовать отключения интернета в отдельных странах. Гибкость создания таких эффектов перекрестного выделения – одна из возможностей D3, которые мне больше всего нравятся в этой библиотеке. Является ли это одной из причин, по которым вы выбрали D3 для этого проекта?

Да, конечно! Правда, здесь основное взаимодействие было реализовано с использованием «хранилищ» Svelte, но логика похожа. Мы хотели дать пользователям возможность исследовать данные все глубже и глубже, поэтому диаграмма предлагает фильтрацию, выделение и прямую анимацию с помощью кнопки **Play** (Запустить).

Какие проекты имеются сейчас в ваших планах на будущее?

После создания представленного проекта нам посчастливилось поработать с Rest of World еще над несколькими проектами, включая недавний проект, демонстрирующий глобальный технологический спад 2022 года. В дальнейшем моя студия надеется продолжить сотрудничество с социально ориентированными организациями и помогать им рассказывать истории с помощью визуализации данных.

Итоги

- Современные веб-интерфейсы обычно создаются с использованием фреймворков, таких как React, Angular или Svelte. Если вы профессионально занимаетесь визуализацией данных с помощью D3, то вам, скорее всего, потребуется интегрировать D3 с одним из этих фреймворков.
- Существует три подхода к использованию D3 с фреймворком: изоляция элемента DOM и передача управления им библиотеке D3, манипулирование DOM только с помощью фреймворка и использование D3 только в роли вспомогательной библиотеки, а также гибридный подход, предполагающий предоставление библиотеке D3 доступа к DOM там, где это дает явную выгоду.

- Фреймворки позволяют манипулировать частями DOM с помощью ссылок. После установки ссылки мы можем использовать D3 так же, как делали это в предыдущих главах.
- При использовании второй стратегии, манипулируя деревом DOM исключительно с помощью фреймворка, мы не можем задействовать шаблон привязки данных D3, генераторы осей, обработчики событий и переходы. Но у нас сохраняется доступ ко множеству полезных методов, таких как шкалы и генераторы форм, которые пригодятся при создании визуализаций на основе фреймворков.
- Примером гибридного подхода может служить сценарий, когда нужно анимировать атрибут `d` элемента `path`. В таких случаях очень удобно использовать метод `transition` из библиотеки D3, не опасаясь, что это окажет существенное влияние на производительность проекта.

9 Адаптивные визуализации

В этой главе рассматриваются:

- основные принципы адаптивного дизайна;
- приемы динамической адаптации размеров текстовых элементов для соответствия экранам меньшего размера;
- особенности использования макета адаптивной сетки;
- адаптация плотности информации к разным размерам экрана;
- изменение ориентации диаграммы.

В веб-дизайне вопрос об обязательной адаптивности веб-сайта или веб-приложения больше не стоит. Мы считаем само собой разумеющимся, что пользователи будут обращаться к нашим проектам с телефонов, планшетов и настольных компьютеров. У нас есть четкие рекомендации и накоплен богатый опыт, а адаптация стандартных цифровых функций к разным размерам экрана – простая задача. Но в визуализации цифровых данных дело обстоит несколько сложнее и часто сильно зависит от типа визуализации и целевой аудитории.

В этой главе мы преобразуем линейный график из главы 4 и информационную панель из главы 8 и сделаем их адаптивными. В процессе обсуждения мы дадим важные советы и представим идеи,

которые пригодятся вам при создании адаптивных цифровых визуализаций с помощью D3.

9.1 Что такое адаптивный дизайн?

Адаптивный дизайн – это дизайн, который адаптируется к наиболее распространенным размерам экранов, от настольного компьютера до планшета и мобильного устройства, и поддерживает возможность взаимодействия не только с мышью, но и посредством сенсорного экрана. По данным Statcounter (https://gs.statcounter.com/), в июле 2023 года на мобильные устройства приходилось 56 % общего интернет-трафика, на настольные компьютеры – 42 %, а на планшеты – 2 % (см. рис. 9.1). Отсюда следует, что, создавая адаптивные визуализации с учетом пользователей мобильных устройств, мы значительно расширяем свою аудиторию. Например, визуализации из новостных статей или историй, опубликованных в социальных сетях, с большой вероятностью будут потребляться на мобильных устройствах.

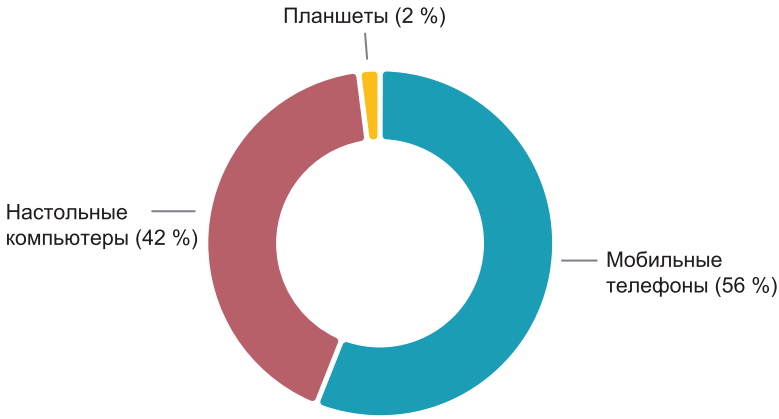


Рис. 9.1 В июле 2023 года на мобильные устройства приходилось 56 % общего интернет-трафика, на настольные компьютеры – 42 % и на планшеты – 2 % (источник: Statcounter)

Методы разработки адаптивных веб-сайтов хорошо известны, но они отличаются от методов создания адаптивных визуализаций данных. Как отмечают Хоффсвелл, Уилмот и Лю[1]:

> «*Методы адаптивного веб-дизайна [...] нельзя напрямую перенести на визуализацию: для достижения в веб-дизайне в первую очередь используются перенос текста, изменение*

[1] Jane Hoffswell, Wilmot Li, and Zhicheng Liu, «Techniques for Flexible Responsive Visualization Design», in *CHI*, 2020.

> *размера изображений и перекомпоновка документа; эти подходы слабо связаны с решением таких проблем визуализации, как представление данных, настройка масштаба или размещение аннотаций*».

Поскольку процесс адаптации очень сложный и может потребовать много времени, крайне важно запланировать дополнительное время и бюджет при разработке адаптивных визуализаций.

При работе с визуализациями цифровых данных важно учитывать два фундаментальных подхода к адаптивному веб-дизайну: ориентация преимущественно на мобильные устройства и ориентация преимущественно на настольные компьютеры. Оба этих подхода мы рассмотрим в следующих подразделах.

9.1.1 Подход, ориентированный на мобильные устройства

В подходе с ориентацией на мобильные устройства мы сначала создаем дизайн мобильной компоновки. Этот процесс часто подразумевает создание более простых визуализаций, адаптированных к портретной ориентации экранов телефонов. После этого мобильный дизайн можно расширить на большие экраны, изменив компоновку и добавив дополнительную информацию и функции.

На этапе реализации сначала создаем мобильную версию и при этом гарантируем, что загружаться будут только те стили и код, которые применяются к мобильной версии, что может повысить производительность. Медиазапросы CSS имеют тип `min-width`, что позволяет постепенно добавлять стили и функции, которые применяются только к большим экранам:

```
@media (min-width: 600px) { ... }
```

9.1.2 Подход, ориентированный на настольные компьютеры

При подходе с ориентацией на настольные компьютеры сначала создается дизайн макета с большим разрешением. Он может быть более сложным и включать множество деталей и взаимодействий. Затем этот дизайн адаптируется для небольших экранов, при этом иногда макет упрощается и фокусируется на передаче самой важной информации.

На этапе реализации сначала создается версия для настольного компьютера. Медиазапросы CSS имеют тип `max-width`, что позволяет постепенно добавлять стили, которые применяются только к небольшим экранам:

```
@media (max-width: 600px) { ... }
```

9.2 Адаптивный линейный график

В этом разделе мы вернемся к графику средней недельной температуры, созданному в главе 4, и сделаем его адаптивным. По мере обсуждения мы представим несколько советов и идей по созданию адаптивных графиков. Адаптивный макет, который мы создадим, можно увидеть на рис. 9.2 и в онлайн-версии проекта по адресу http://mng.bz/5orB.

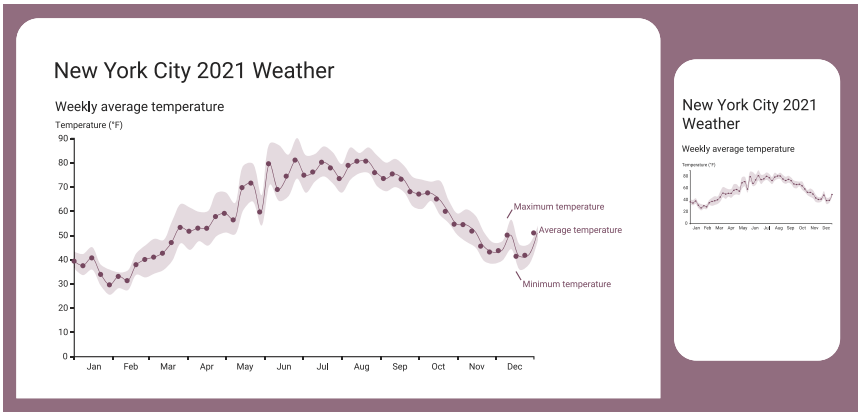


Рис. 9.2 Адаптивный линейный график

Сначала откройте папку `9.2-Responsive_line_chart/start` в редакторе и запустите локальный веб-сервер. Обратитесь к приложению A, если вы забыли, как действовать дальше.

> **ПРИМЕЧАНИЕ** Файл с кодом для этой главы можно найти в репозитории книги на GitHub по адресу http://mng.bz/Xqjv.

Откройте проект в браузере и поэкспериментируйте с размером окна. Мы советуем открыть также инструмент инспектора и закрепить его справа вверху на экране, поскольку изменить размер инспектора проще, чем всего экрана. Кроме того, в браузерах Chrome и Firefox есть возможность имитировать мобильные устройства.

Мы уже проделали большую работу, чтобы сделать нашу диаграмму адаптивной, установив атрибут `viewBox` контейнера SVG и избавившись от необходимости устанавливать его атрибуты `width` и `height`. Эта стратегия подробно обсуждалась в главе 1, разделе 1.1.5. Но, попробовав отобразить диаграмму на небольших экранах, вы могли заметить несколько проблем на экранах небольшого размера (рис. 9.3). Первая проблема – метки становятся слишком маленькими и нечитабельными. Это обусловлено тем, что текст SVG действует подобно любому другому графическому элементу и уменьшается вместе

с остальной частью диаграммы. Вторая проблема – недостаток горизонтального пространства для правильного распределения линейной диаграммы. Точки и метки осей оказываются расположенными слишком близко друг к другу.

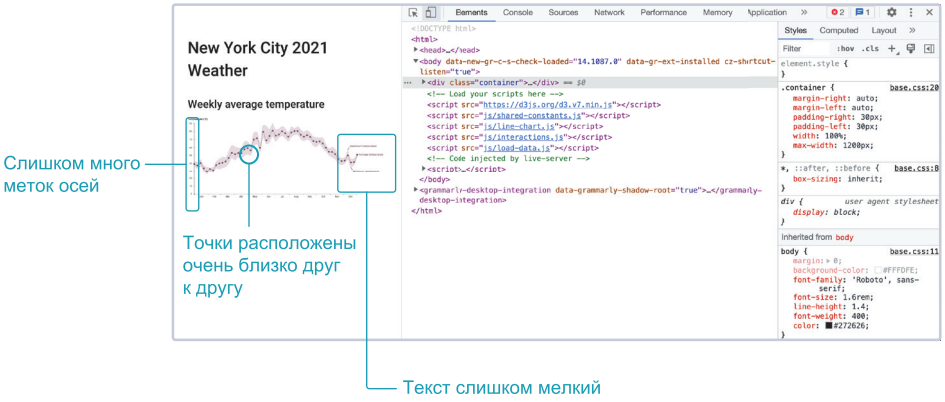


Рис. 9.3 Несмотря на адаптивность, благодаря применению атрибута viewBox к контейнеру SVG в визуализации все еще наблюдается несколько проблем, которые необходимо решить: текстовые метки слишком малы, очень много меток осей, а точки располагаются чересчур близко друг к другу

При изменении визуализаций для придания им адаптивности следует учесть два аспекта:

1. *размер экрана, с которым проект отображается в первый момент*. При открытии визуализации на любом устройстве – телефоне, планшете или настольном компьютере – она должна быть читаемой и адаптированной к разрешению экрана;
2. *пользователь может изменить размер экрана*. На настольных компьютерах пользователи могут изменить размер окна браузера, а на телефонах и планшетах – ориентацию устройства. Визуализация должна динамически адаптироваться к этим изменениям с помощью медиазапросов CSS или обрабатывая событие изменения размера окна в JavaScript.

9.2.1 Адаптация размера текстовых надписей

Для начала рассмотрим проблему с размером текстовых меток. В настоящее время метки осей имеют размер 14 пикселей, тогда как другие – 16 пикселей. Контейнер SVG линейной диаграммы имеет атрибут `viewBox` со значениями `0, 0, 1200, 500`. На настольных компьютерах, где контейнер SVG может распахиваться до полного размера (1200×500 пикселей), метки соответствуют своему CSS-свойству `font-size` и легко читаются. Но с уменьшением ширины окна графи-

ка SVG тоже уменьшается, в том числе и текстовые элементы, что делает их нечитаемыми на экранах телефонов.

Чтобы исправить эту проблему, нужно использовать увеличенный размер шрифта на меньших экранах. Для этого воспользуемся приемом, которым поделилась Надия Бремер в своей статье «Techniques for Data Visualization on Both Mobile & Desktop» (www.visualcinnamon.com/2019/04/mobile-vs-desktop-dataviz/), основанном на применении линейной шкалы для расчета размера шрифта исходя из ширины экрана.

Откройте файл `shared-constants.js` и объявите следующую шкалу `fontSizeScale`. Ее предметная область, ширина экрана, простирается от 600 до 1000 пикселей, а выходной диапазон, размер шрифта, – от 26 до 16 пикселей. Обратите внимание, что мы также включаем свойство `clamp()`, чтобы D3 не экстраполировала значения за пределы диапазона. Для экранов с шириной 600 пикселей или меньше размер шрифта останется равным 26 пикселям, а для экранов с шириной 1000 пикселей или больше размер шрифта останется равным 6 пикселям. Внутри интервала [600…1000] размер шрифта будет меняться:

```
const fontSizeScale = d3.scaleLinear()
  .domain([600, 1000])
  .range([26, 16])
  .clamp(true);
```

Чтобы применить эту шкалу к визуализации, нужно знать ширину экрана. В начале файла `shared-constants.js` добавьте следующий фрагмент. Он объявляет функцию `getWindowWidth()`, которая возвращает свойство `innerWidth` JavaScript-объекта `window`, а затем вызывает ее и сохраняет результат в константе `windowWidth`:

```
const getWindowWidth = () => {
  return window.innerWidth;
};
let windowWidth = getWindowWidth();
```

В файле `line-chart.js`, в конце функции `drawLineChart()`, вызовите `fontSizeScale`, чтобы задать стилевое свойство `font-size` каждого текстового элемента, присутствующего в `div` с идентификатором `"line-chart"`:

```
d3.selectAll("#line-chart text")
  .style("font-size", `${fontSizeScale(windowWidth)}px`);
```

Однако эта стратегия адаптирует размер шрифта только один раз: при загрузке страницы. Если изменить размер экрана уже после загрузки страницы, то ничего не произойдет, и текстовые метки попрежнему будут иметь неоптимальный размер. Чтобы исправить эту проблему, откройте файл `resize.js` и добавьте обработчик события

resize, возникающего при изменении размера окна. В обработчике события resize обновите значение windowWidth и вызовите функцию resizeChart(), которая обновит размер шрифта каждого текстового элемента с помощью fontSizeScale:

```
const resizeChart = () => {
  d3.selectAll("#line-chart text")
    .style("font-size", `${fontSizeScale(
      windowWidth)}px`);
};
```

Адаптировать размер шрифта каждого текстового элемента в диаграмме с помощью fontSizeScale

```
window.addEventListener("resize", () => {
  windowWidth = getWindowWidth();
  resizeChart();
});
```

Добавить обработчик события изменения размера окна. При получении события обновить значение windowWidth и вызвать функцию resizeChart()

Наконец, чтобы избежать дублирования кода, вернитесь к функции drawLineChart() в line-chart.js и вместо непосредственного обновления свойства font-size вызовите resizeChart(). Теперь resizeChart() будет выполняться как при загрузке страницы, так и при изменении размера экрана.

Откройте в браузере инструмент инспектора и выберите одну из текстовых меток на диаграмме. Попробуйте изменять размер окна. Вы должны увидеть, что размер шрифта динамически обновляется!

9.2.2 Настройка меток осей

Поскольку метки на экранах телефонов стали больше, можно утверждать, что на оси y слишком много меток. В настоящее время меткой отмечаются каждые 10 °F. Давайте посмотрим, как динамически уменьшить их количество.

Прежде всего разберемся с распознаванием макетов для настольных компьютеров и для мобильных устройств. В файле shared-constants.js объявите новую константу с именем isDesktopLayout. Если ширина окна больше или равна 700 пикселям, присвойте константе значение true, в противном случае – false:

```
let isDesktopLayout = windowWidth >= 700 ? true : false;
```

Затем в line-chart.js укажите количество рисок на оси y, исходя из текущего макета. На компьютере пусть будет видно 10 делений, а на мобильном устройстве – 5:

```
leftAxis = d3.axisLeft(yScale)
  .ticks(isDesktopLayout ? 10 : 5);
```

Наконец, для обработки изменения размера окна вернитесь в resize.js. Чтобы количество рисок обновлялось только при пересече-

нии границы 700 пикселей, когда происходит смена макета, добавьте условие внутри `resizeChart()`. Если ширина окна больше или равна 700 пикселей и `isDesktopLayout` имеет значение `false`, или если ширина окна меньше 700 пикселей и `isDesktopLayout` имеет значение `true`, то обновите значение `isDesktopLayout`.

Затем вызовите генератор `leftAxis`, который мы сделали доступным, объявив его внутри `shared-constant.js`, и обновите количество рисок. Потом выберите группу, содержащую элементы оси y (с классом `"axis-y"`), и вызовите `leftAxis`, применив плавный переход:

```
const resizeChart = () => {
  ...
  if ((windowWidth >= 700 && !isDesktopLayout) ||
      (windowWidth < 700 && isDesktopLayout)) {

    isDesktopLayout = !isDesktopLayout;

    leftAxis
      .ticks(isDesktopLayout ? 10 : 5);
    d3.select(".axis-y")
      .transition()
        .call(leftAxis);
  }
};
```

Если ширина окна больше или равна 700 пикселей и isDesktopLayout имеет значение false, или если ширина окна меньше 700 пикселей и isDesktopLayout имеет значение true, то выполнить тело условной инструкции

Обновить значение isDesktopLayout

Обновить количество рисок на leftAxis

Вызвать генератор leftAxis, чтобы обновить ось y

Теперь количество рисок на левой оси должно обновляться динамически!

9.2.3 Использование минималистского подхода

Работа с маленькими экранами часто означает использование минималистского подхода. Мы фокусируемся на критических фрагментах информации и удаляем все второстепенное.

Примером могут служить аннотации, отображаемые справа. Они полезны, но не обязательны для понимания диаграммы. Кроме того, они занимают много горизонтального пространства, которого и так не хватает на телефонах (особенно в портретной ориентации).

В конце файла `line-chart.js` мы уже вызываем функцию `appendAnnotations()`, которая находится в `annotations.js`. Давайте будем вызывать эту функцию, только если константа `isDesktopLayout` имеет значение `true`. Данное условие гарантирует, что эти аннотации не будут отображаться на экране телефона:

```
if (isDesktopLayout) {
  appendAnnotations();
}
```

В `shared-constants.js` установите также ширину правого отступа, исходя из текущего макета. В макете для настольного компьютера правый отступ должен иметь ширину 200 пикселей, а в макете для мобильных устройств – 10 пикселей:

```
const margin = {
  top: 35,
  right: isDesktopLayout ? 200 : 10,
  bottom: 35,
  left: 45
};
```

Затем добавьте в `annotations.js` новую функцию с именем `removeAnnotations()`. В этой функции с помощью D3 выберите группу с классом `".annotations"` и удалите ее из DOM вызовом метода `remove()`:

```
const removeAnnotations = () => {
  d3.select(".annotations").remove();
};
```

Наконец, вернитесь к функции `resizeChart()` внутри `resize.js`. В условии, где обновляется `isDesktopLayout`, пересчитайте правый отступ и внутреннюю ширину диаграммы. Затем, как показано в листинге 9.1, обновите диапазон `xScale`, нижнюю ось и элементы диаграммы, а также добавьте или удалите аннотации.

Листинг 9.1 Обновление кривой и правого отступа (resize.js)

```
const resizeChart = () => {

  ...

  if ((windowWidth >= 700 && !isDesktopLayout) ||
      (windowWidth < 700 && isDesktopLayout)) {

    ...

    margin.right = isDesktopLayout ? 250 : 10;
    innerWidth = width - margin.left - margin.right;

    xScale.range([0, innerWidth]);

    bottomAxis = d3.axisBottom(xScale)
      .tickFormat(d3.timeFormat("%b"));
    d3.select(".axis-x")
      .transition()
        .call(bottomAxis);
    positionXaxisLabels();
```

Обновить правый отступ и ширину внутренней диаграммы

Обновить диапазон xScale

Обновить нижнюю ось и пересчитать местоположения рисок осей

```
    if (isDesktopLayout) {
      appendAnnotations();
    } else {
      removeAnnotations();
    }

    d3.select(".temperature-area")
      .transition()
        .attr("d", areaGenerator(data));

    d3.select(".temperature-curve")
      .transition()
        .attr("d", curveGenerator(data));

    d3.selectAll("circle")
      .data(data)
      .join("circle")
      .transition()
        .attr("r", 5)
        .attr("cx", d => xScale(d.date))
        .attr("cy", d => yScale(d.avg_temp_F))
        .attr("fill", aubergine);

  }

};
```

Если используется макет для компьютера, то вызвать функцию, которая добавляет аннотации; иначе удалить аннотации

Обновить область графика температуры, кривую и точки данных

По завершении линейная диаграмма должна существенно лучше адаптироваться к экранам мобильных устройств (рис. 9.4). Как вы могли заметить, мы внесли лишь ряд незначительных изменений: увеличили размер меток, скорректировали количество рисок на осях и удалили вторичную информацию. Но если сравнить исходный проект (http://mng.bz/46dw) с окончательным (http://mng.bz/wxBg) на реальном телефоне, то увидите, что теперь диаграмма читается значительно лучше. В следующем разделе мы рассмотрим адаптацию более сложного проекта.

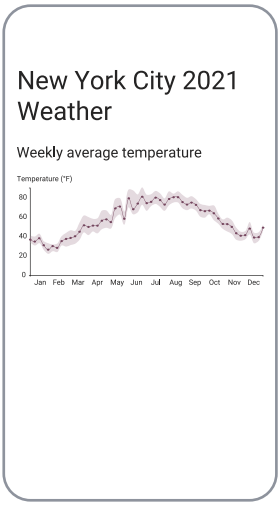


Рис. 9.4 Законченная адаптивная линейная диаграмма

9.3 Адаптивная информационная панель

В главе 8 мы создали информационную панель, построенную с помощью React, где представили сравнительные характеристики нескольких популярных фреймворков JavaScript. В данном разделе мы

сделаем эту панель адаптивной, как показано на рис. 9.5, и познакомимся с некоторыми дополнительными приемами адаптации диаграмм.

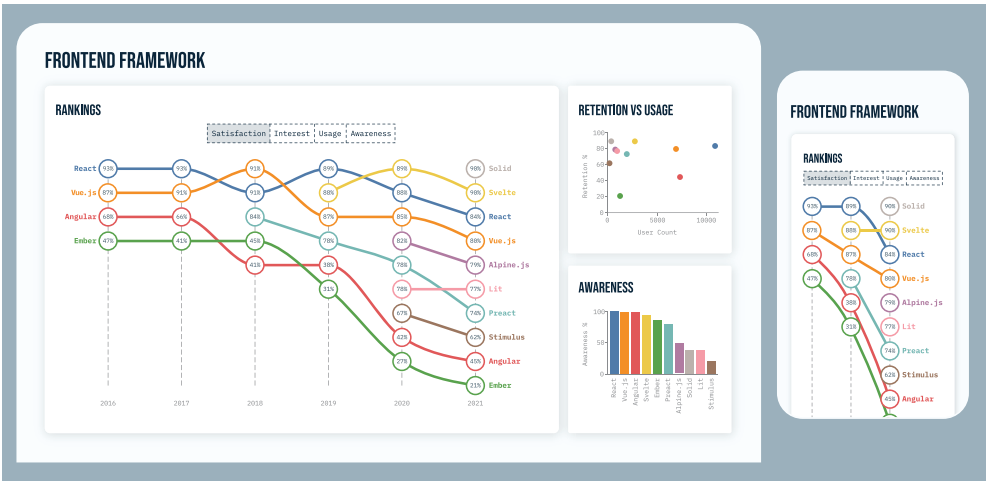


Рис. 9.5 Адаптивная информационная панель

Поскольку информационная панель создана с помощью React, вы должны иметь некоторое знакомство с этим фреймворком, чтобы следовать нашим инструкциям. В частности, вы должны знать, как создавать компоненты, как передавать свойства дочерним и родительским компонентам, как управлять состоянием компонента и как использовать ловушки React. Но даже если вы незнакомы с React или не хотите его использовать, то все равно прочитайте этот раздел, потому что здесь мы поделимся советами по организации адаптивного дизайна, которые применимы к любому цифровому проекту.

9.3.1 Использование адаптивной сетки

Адаптивная сетка CSS – отличное начало для работы с более сложным макетом, который может содержать несколько визуализаций или смесь визуализаций и текста. Например, в нашей информационной панели используется сетка с 12 колонками, подобная системе сеток Bootstrap. С помощью медиазапросов CSS и стратегии на основе имен классов мы можем создать три макета для отображения нашей информационной панели: один для настольного компьютера, один для планшета и один для мобильного устройства. Каждый прямоугольник на рис. 9.6 содержит визуализацию в следующем порядке: диаграмма рейтингов, диаграмма рассеяния и столбчатая диаграмма.

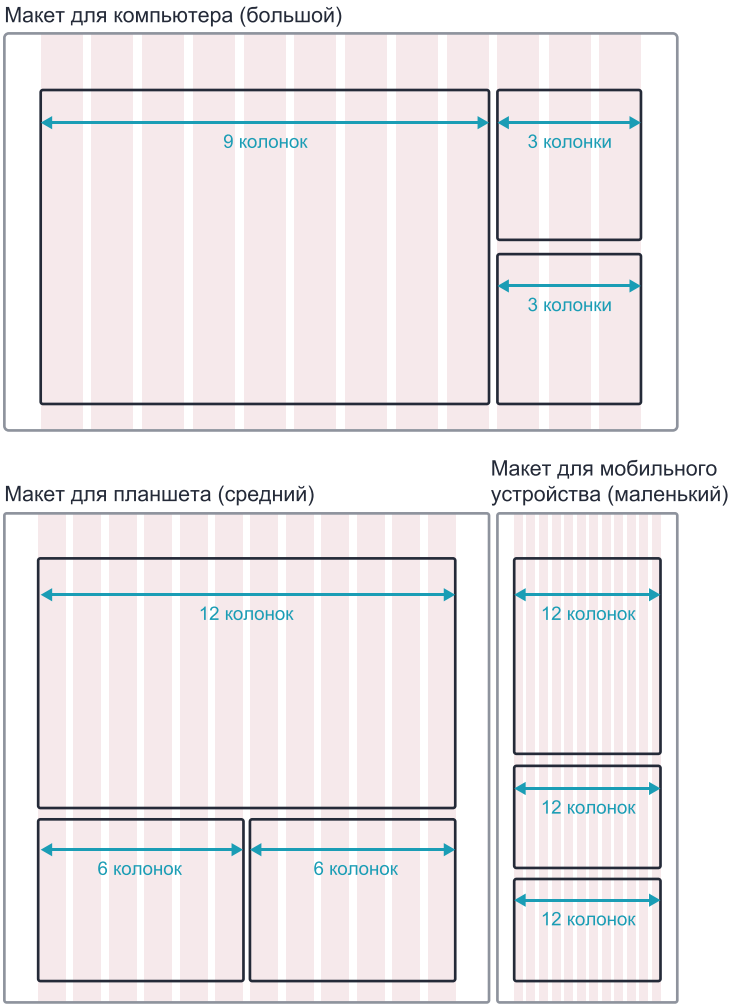


Рис. 9.6 Использование адаптивной сетки для позиционирования визуализаций в зависимости от ширины экрана

Адаптивная сетка CSS flexbox включает компоненты трех типов: контейнер, ряды и колонки. Контейнер отвечает за центрирование контента на странице, обеспечивает горизонтальный отступ и гарантирует, что ширина контента не превысит указанную максимальную ширину. Перейдя в папку `start` в каталоге `9.3-Responsive_dashboard` и открыв файл `grid.css`, вы найдете следующее объявление стиля для элемента с классом `container`. Свойствам `margin-right` и `margin-left` присвоено значение `auto`, что обеспечивает центрирование `div`. С обеих сторон также добавлены поля шириной 20 пикселей, а свойству `max-width` присвоено значение `1400px`:

```
.container {
  margin-right: auto;
  margin-left: auto;
  padding-right: 20px;
  padding-left: 20px;
  width: 100%;
  max-width: 1400px;
}
```

На рис. 9.7 контейнер `div` изображен в виде зеленого прямоугольника. Как вы видите, он охватывает все содержимое панели.

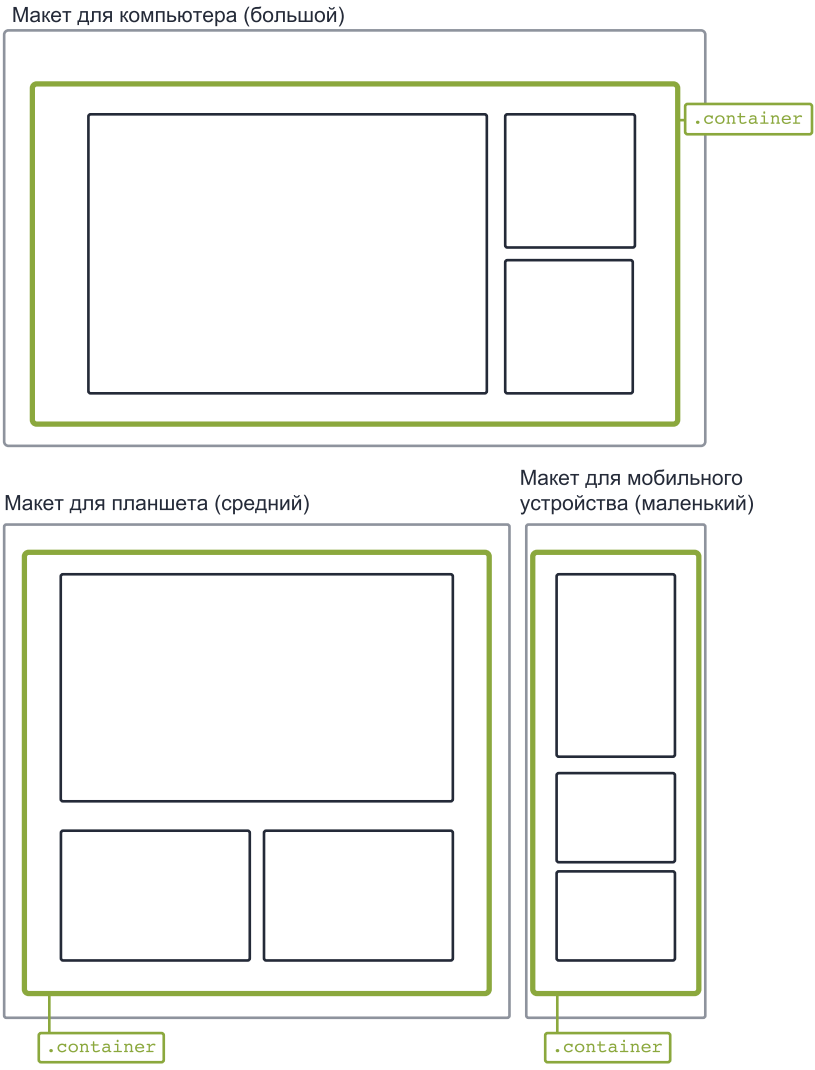


Рис. 9.7 Контейнер отвечает за центрирование контента на странице, а также чтобы ширина контента не превысила указанную максимальную ширину. Он вмещает все визуализации и текст

Горизонтальные блоки контента заключены в ряды. Для создания адаптивного макета нам понадобятся два `div` с классом `row`, показанные на рис. 9.8 в виде желтых прямоугольников. В макете для настольного компьютера внешний ряд (`div` с классом `row`) создает блок, содержащий визуализацию рейтингов слева и диаграмму рассеяния со столбчатой диаграммой справа. В макете для планшета внутрен-

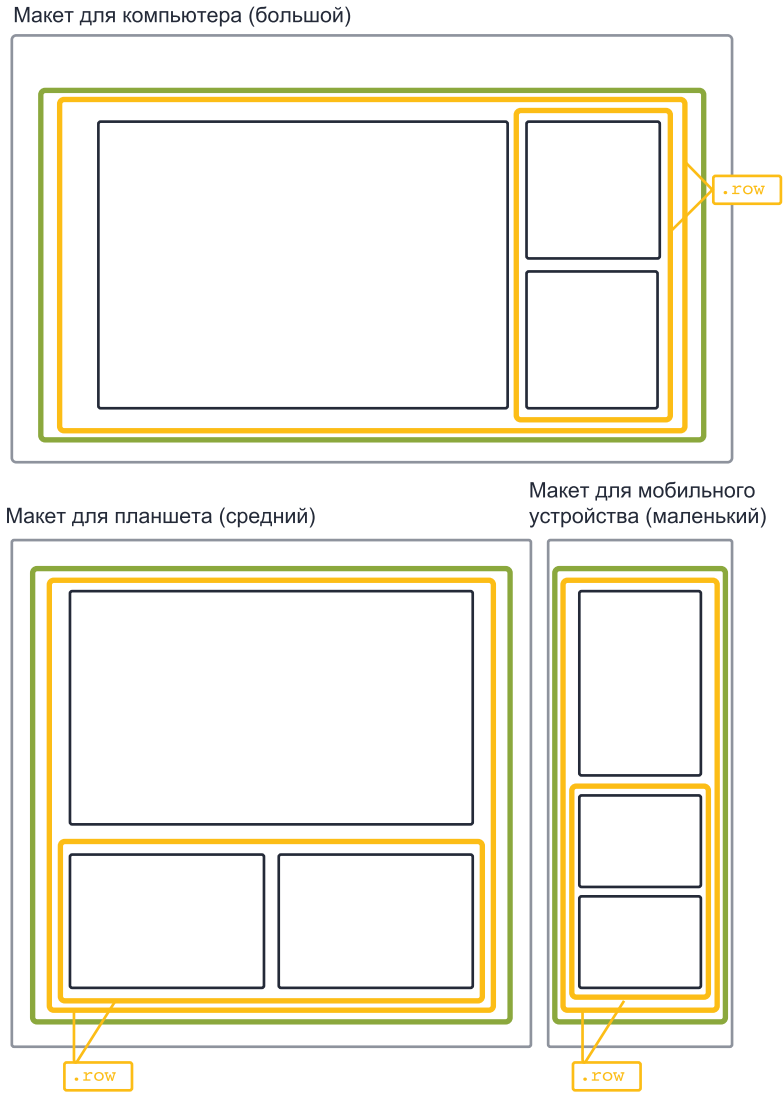


Рис. 9.8 Ряды (элементы div с классом row) заключают горизонтальные блоки контента. Для создания адаптивного макета нужны два ряда: один для настольного компьютера с диаграммой рейтинга слева и другими диаграммами справа и один для планшета со вторым горизонтальным блоком, включающим диаграмму рассеяния и столбчатую диаграмму

ний ряд создает второй горизонтальный блок, где диаграмма рассеяния отображается слева, а столбчатая диаграмма – справа. Оба ряда присутствуют на экранах всех размеров.

В следующем фрагменте мы присваиваем свойству `display` рядов значение `flex`, что позволит без особых трудностей сгенерировать желаемый макет. Свойство `flex-wrap` гарантирует возможность переноса дочерних элементов и их расположение в несколько рядов, а отрицательные поля `margin-left` и `margin-right` учитывают зазор или горизонтальное пространство между элементами, которое будет создано с помощью колонок:

```
.row {
  display: flex;
  flex-wrap: wrap;
  margin-left: -10px;
  margin-right: -10px;
}
```

В этой системе сеток слово «колонка» имеет два значения, что может сбивать с толку. Во-первых, *колонки* – это элементы `div` в разметке с классом, имя которого начинается с `"col-"`. Во-вторых, слово «колонка» используется в названии *сетка с 12 колонками*, которое обозначает дизайн для горизонтального распределения нашего контента.

На рис. 9.9 показаны колонки как элементы `div`, в виде синих прямоугольников. Они определяют ширину своего содержимого и должны быть непосредственными потомками элементов рядов, чтобы обеспечить правильную работу стилей.

Макет с колонками настраивается с помощью двух наборов свойств. Первый применяется ко всем элементам с классом, имя которого начинается с `"col-"`. Такие элементы имеют левое и правое поля с размером 10 пикселей, создающие зазор, и ширину, составляющую 100 % от ширины родителя:

```
[class^="col-"] {
  position: relative;
  width: 100%;
  min-height: 1px;
  padding-right: 10px;
  padding-left: 10px;
}
```

Как упоминалось выше, наш дизайн основан на сетке с 12 колонками. Такие сетки широко используются на практике, потому что число 12 легко делится на 12, 6, 4, 3, 2 и 1. У нас также есть три размера экрана: маленький, средний и большой. Если использовать подход с ориентацией преимущественно на мобильные устройства, то

сначала нужно дать имена классов, которые применяются к маленьким, затем к средним и, наконец, к большим экранам.

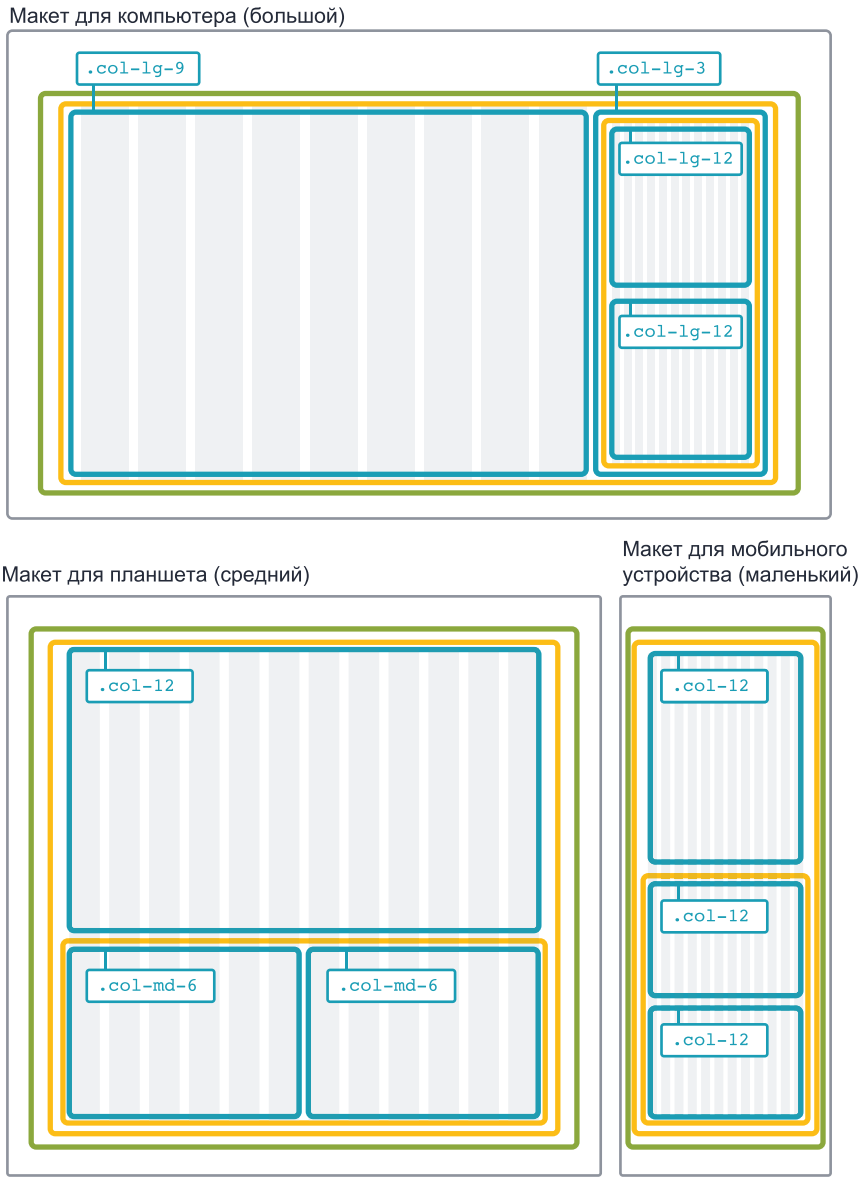


Рис. 9.9 Элементы колонок определяют ширину своего содержимого. Они должны быть непосредственными потомками элементов рядов

Давайте возьмем в качестве примеров диаграмму рассеяния и столбчатую диаграмму. На маленьких экранах (например, на

экранах телефонов) желательно, чтобы каждая занимала всю ширину экрана. Мы присвоим им класс `col-12`, чтобы они заняли все 12 колонок в ширину. На средних экранах (например, планшетах) желательно, чтобы они заняли 50 % ширины экрана, или 6 колонок, поэтому им будут присвоены имена классов `col-md-6`. Это все равно что сказать: «на экранах средней ширины диаграммы должны занять шесть колонок». Наконец, на больших экранах (например, настольных компьютерах) желательно, чтобы они заняли всю доступную ширину в родительском ряду (12 колонок). Мы присвоим им имя класса `col-lg-12`, что означает: «на больших экранах диаграммы должны занять 12 колонок». Обратите внимание, что каждый элемент ряда создает новую сетку из 12 колонок. В макете для компьютера внешний ряд охватывает диаграмму рейтинга, которая занимает 9 колонок, и правую часть экрана, которая занимает 3 колонки. Правая часть содержит второй элемент ряда, с которого начинается новый макет из 12 колонок. Диаграмма рассеяния и столбчатая диаграмма, которые он содержит, занимают в ширину все 12 колонок в этом элементе.

Следующий фрагмент наглядно показывает, как используются эти имена классов. Для случая с 12 колонками с помощью сокращенной записи `flex` мы присваиваем значения `0 0 100%` свойствам `flex-grow`, `flex-shrink` и `flex-basis` соответственно, а также устанавливаем свойство `max-width` `100%`. Свойствам `flex-grow` и `flex-shrink` мы присваиваем `0`, чтобы гарантировать соблюдение требуемой ширины колонок, а `flex-basis` обрабатывает фактическую ширину столбца.

Также различные размеры экрана обрабатываются с помощью медиазапросов CSS: средний экран имеет `min-width 600px`, а большой – `min-width 1100px`. Эти числа будут варьироваться от проекта к проекту и выбираются так, чтобы информационная панель выглядела хорошо на экранах любого размера:

```
.col-12 {
  flex: 0 0 100%;
  max-width: 100%;
}
.col-3 {
  flex: 0 0 25%;
  max-width: 25%;
}
@media (min-width: 600px) {
  .col-md-6 {
    flex: 0 0 50%;
    max-width: 50%;
  }
}
```

```
@media (min-width: 1100px) {
  .col-lg-12 {
    flex: 0 0 100%;
    max-width: 100%;
  }
}
```

Если вы не имеете опыта использования CSS-свойства `flex`, то вам могут потребоваться дополнительные пояснения, которые можно найти, например, в статье «A Complete Guide to Flexbox» Криса Койера (Chris Coyier; http://mng.bz/qOMz).

> **ПРИМЕЧАНИЕ** Адаптивную сетку можно также создать с помощью CSS-свойства `grid`.

Давайте посмотрим, как эта стратегия воздействует на нашу информационную панель. Откройте проект в редакторе, откройте терминал и выполните команду `npm install`, а затем `npm run start`.

В браузере вы увидите, что для отображения визуализации используется макет для компьютера, делящий пространство окна по ширине на 9 и 3 колонки. Но изменение размера экрана не приводит к смене макета. В файле `App.js` у нас уже есть `div` с классом `container`. В `Charts.js` у нас есть два `div` с классом `row`, как показано на рис. 9.7. Поэтому нам нужно только изменить имена классов, применяемых к колонкам диаграммы (элементы `div` с классами, начинающимися с `col-`).

Мы присваиваем обертке диаграммы рейтингов имена классов `col-12` и `col-lg-9`. Эти классы означают: «на маленьких экранах занимают все 12 колонок, а на больших – 9». Аналогично элементу `div`, который вмещает диаграмму рассеяния и столбчатую диаграмму, мы присваиваем классы `col-12` и `col-lg-3`. Наконец, отдельные обертки для диаграммы рассеяния и столбчатой диаграммы получают имена классов `col-12`, `col-md-6` и `col-lg-12`. На маленьких экранах они будут распределяться по 12, на средних экранах – по 6 и на больших экранах – по 12 колонкам или на всю ширину родительского элемента с классом `row`.

Листинг 9.2 Создание адаптивного макета (Charts.js)

```
<Fragment>
  <h1>Front-end Frameworks</h1>
  <div className='row'>
    <div className='col-12 col-lg-9'>
      <Rankings ... />
    </div>
    <div className='col-12 col-lg-3'>
      <div className='row'>
```

На маленьких экранах элемент занимает все 12 столбцов. На больших экранах – 9

На маленьких экранах элемент занимает все 12 столбцов. На больших экранах – 3

```
          <div className='col-12 col-md-6 col-lg-12'>
            <ScatterplotReactControlled ... />
          </div>
          <div className='col-12 col-md-6 col-lg-12'>
            <BarChart ... />
          </div>
        </div>
      </div>
    </div>
    ...
</Fragment>
```

На маленьких экранах элемент занимает все 12 столбцов, на средних экранах – 6 и на больших экранах – 12

Поскольку все необходимые стили уже объявлены в `grid.css`, то при изменении размера экрана информационная панель должна адаптироваться к трем макетам, показанным на рис. 9.6–9.9.

9.3.2 Адаптация плотности информации

Теперь, разобравшись с адаптивной сеткой, сосредоточимся на таблице рейтингов. На маленьких экранах значки становятся крошечными, а метки трудно читаются. Поэтому при отображении на мобильных устройствах желательно уменьшить плотность информации, сосредоточившись только на самом важном. Например, можно показывать рейтинги не за все годы с 2016 по 2021, а только за 2016, 2019 и 2021. Наша история все равно останется правдивой, но будет сосредоточена на основных изменениях, происшедших между этими годами.

Прежде всего нам нужно узнать текущий макет. Допустим, мы решили выбирать макет для компьютеров, если экран имеет ширину 600 пикселей или больше, и мобильный макет на экранах меньшего размера. В листинге 9.3 мы определяем константу `breakpoint` и присваиваем ей значение `600`. Затем объявляем функцию `getLayout()`, которая возвращает `"desktop"` или `"mobile"` в зависимости от значения свойства `innerWidth` JavaScript-объекта `window`. Обратите внимание, константа и функция объявлены вне компонента `Charts`, потому что им не нужны никакие свойства или переменные компонента. Затем внутри компонента `Charts` объявляем переменную состояния с именем `layout` и инициализируем ее значение вызовом `getLayout()`.

Внутри анонимной функции, передаваемой в ловушку `useEffect`, мы подключаем обработчик событий к объекту `window`. Теперь при каждом изменении размеров окна будет вызываться функция `handleWindowResize()` и получать `innerWidth` окна. Если ширина больше или равна значению `breakpoint` и выбран макет `"mobile"`, или если ширина меньше значения `breakpoint` и выбран макет `"desktop"`, то мы обновляем переменную состояния `layout`, вызывая `getLayout()`.

Обратите внимание, что в операторе `return` в ловушке `useEffect` мы также определяем функцию очистки, которая открепляет обработчик событий при размонтировании компонента. Мы также передаем переменную состояния `layout` как зависимость.

Листинг 9.3 Обработка события изменения размера окна (Charts.js)

```
import { Fragment, useState, useEffect } from 'react';
import * as d3 from 'd3';

const breakPoint = 600;
const getLayout = () => {
  const layout = window.innerWidth >= breakPoint
    ? "desktop" : "mobile";
  return layout;
};

const Charts = props => {
  const [layout, setLayout] = useState(getLayout());
  const margin = {top: 30, right: 10, bottom: 50, left: 60};

  useEffect(() => {

    const handleWindowResize = () => {
      const windowWidth = window.innerWidth;
      if ((windowWidth >= breakPoint && layout ===
        "mobile") || (windowWidth < breakPoint &&
        layout === "desktop")) {
        setLayout(getLayout());
      }
    };

    window.addEventListener('resize',
      handleWindowResize);

    return () => {
      window.removeEventListener('resize',
        handleWindowResize);
    };

  }, [layout]);
  return (
    <Fragment>
      ...
    </Fragment>
  )
};

export default Charts;
```

Импортировать ловушки useState и useEffect

Объявить константу с именем breakpoint и значением 600. Объявить функцию getLayout(), которая возвращает "desktop", если значение свойства innerWidth объекта window больше или равно 600, и "mobile", если меньше

Объявить переменную состояния layout и инициализировать ее вызовом функции getLayout()

Внутри ловушки useEffect подключить обработчик событий к объекту window. При изменении размера окна вызвать функцию handleWindowResize(), которая сравнит innerWidth объекта window с текущим макетом и при необходимости обновит переменную layout

Добавить функцию очистки, которая открепляет обработчик событий при отмонтировании компонента

Добавить переменную состояния layout в зависимости от ловушки useEffect

Далее мы создадим второй компонент рейтинга с именем `RankingsMobile`, который используем для отображения мобильной версии

диаграммы рейтинга. В листинге 9.4 если переменная состояния `layout` имеет значение `"desktop"`, то мы вызываем компонент `Rankings`, в противном случае – `RankingsMobile`. Обратите внимание, что мы переместили массив `rankingFilters` в компонент `Charts`, чтобы сделать его доступным как в `Rankings`, так и в `RankingsMobile`. Также в Charts мы переместили функции, которые обрабатывают выбор фильтра.

Листинг 9.4 Вызов Rankings (Charts.js)

```
...
const rankingFilters = [
  { id: "satisfaction", label: "Satisfaction" },
  { id: "interest", label: "Interest" },
  { id: "usage", label: "Usage" },
  { id: "awareness", label: "Awareness" },
];
```

Перенести объявление массива rankingFilters из Rankings.js в Charts.js

```
const Charts = props => {
  ...

  const [activeFilter, setActiveFilter] =
    useState("satisfaction");
  const filterSelectionHandler = (id) => {
    if (activeFilter !== id) {
      setActiveFilter(id);
    }
  };
```

Перенести обработку переменной состояния activeFilter из Rankings.js в Charts.js

```
  return (
    <Fragment>
      <h1>Front-end Frameworks</h1>
      <div className='row'>
        <div className='col-12 col-lg-9'>
          {layout === "desktop"
            ? <Rankings
                margin={margin}
                data={props.data}
                colorScale={colorScale}
                rankingFilters={rankingFilters}
                activeFilter={activeFilter}
                onFilterSelection={
                 filterSelectionHandler}
              />
            : <RankingsMobile
                margin={margin}
                data={props.data}
                colorScale={colorScale}
                rankingFilters={rankingFilters}
                activeFilter={activeFilter}
                onFilterSelection={
                  filterSelectionHandler}
```

Если переменная состояния layout имеет значение "desktop", то отобразить компонент Rankings, иначе – компонент RankingsMobile. Передать переменные margin, data, colorScale и filter-related как свойства

```
              />
            }
          </div>
          <div className='col-12 col-lg-3'>
            ...
          </div>
        </div>
        ...
      </Fragment>
    )
};

export default Charts;
```

Наконец, в листинге 9.5 приводится код, создающий компонент `RankingsMobile`. Он очень похож на компонент `Rankings`, созданный в предыдущей главе, но имеет ряд отличий:

- другие ширина, высота, правый и левый отступы;
- создается массив с годами, которые будут отображаться в визуализации. Мы выбираем из набора данных первый, последний и срединный годы. Отображение только первого и последнего годов тоже может оказаться хорошим выбором;
- данные об опыте использования, полученные в виде входных свойств, копируются и сохраняются в константе `mobileData`. Затем выполняется обход содержимого `mobileData` и сохраняются только данные, включенные в массив `years`;
- визуализация производится точно так же, как в `Rankings.js`, с той лишь разницей, что метки с названиями фреймворков не отображаются слева от диаграммы.

Листинг 9.5 Создание компонента RankingsMobile (RankingsMobile.js)

```
...
const RankingsMobile = props => {
  const width = 300;
  const height = 550;
  const marginRight = 120;
  const marginLeft = 30;
  const innerWidth = width - marginRight - marginLeft;
  const innerHeight = height - props.margin.top - props.margin.bottom;

  const firstYear = props.data.years[0];
  const lastYear = props.data.years[props.data.
    years.length - 1];
  const middleYear = Math.round(d3.median([firstYear,
    lastYear]));
  const years = [firstYear, middleYear, lastYear];
```

Изменить ширину, высоту, правый и левый отступы диаграммы

Создать массив years с годами для отображения на диаграмме: первый, последний и срединный годы из набора данных

```
  const mobileData = JSON.parse(JSON.stringify(
    props.data.experience));
  mobileData.forEach(framework => {
    framework.awareness = framework.awareness.
      filter(d =>
      ➥ years.includes(d.year));
    framework.interest = framework.interest.filter(d =>
      ➥ years.includes(d.year));
    framework.satisfaction = framework.satisfaction.
      filter(d =>
      ➥ years.includes(d.year));
    framework.usage = framework.usage.filter(d =>
    ➥ years.includes(d.year));
  });
```

Скопировать данные об опыте, полученные в виде входных свойств, и сохранить в константе с именем mobileData. Выполнить обход содержимого mobileData и сохранить только данные, включенные в массив years

```
  const xScale = d3.scalePoint()
    .domain(years)
    .range([0, innerWidth]);
  const yScale = d3.scalePoint()
    .domain(d3.range(1, props.data.ids.length + 1))
    .range([0, innerHeight]);

  return (
    <Card>
      <h2>Rankings</h2>
      <RankingFilters
        filters={props.rankingFilters}
        activeFilter={props.activeFilter}
        onFilterSelection={props.onFilterSelection}
      />
      <ChartContainer
        width={width}
        height={height}
        margin={{ top: props.margin.top, right: marginRight, bottom:
        ➥   props.margin.bottom, left: marginLeft }}
      >
        ...
      </ChartContainer>
    </Card>
  );
};

export default RankingsMobile;
```

По завершении мобильная версия визуализации рейтинга должна выглядеть, как показано на рис. 9.10.

Макет для компьютера и планшета

Макет для мобильного устройства

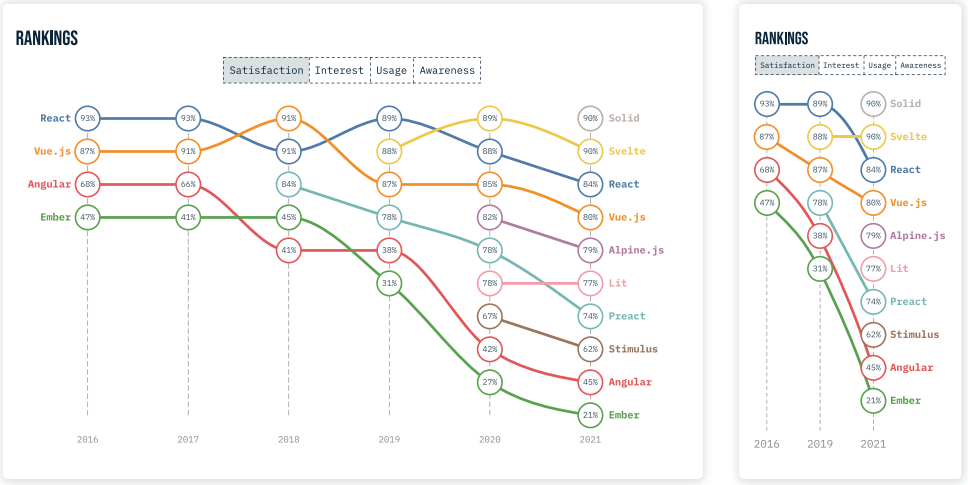


Рис. 9.10 В мобильной версии диаграммы мы сократили количество отображаемых лет. Уменьшение плотности информации позволяет адаптироваться к ширине экрана не в ущерб отображению основных тенденций. Мы также удалили метки слева от диаграммы

9.3.3 *Изменение ориентации диаграммы*

Некоторые типы визуализаций занимают много горизонтального пространства, например линейные диаграммы и потоковые графики. Но телефоны в основном используются в режиме портретной ориентации, когда высота намного больше ширины. Изменение ориентации диаграммы на альбомную могло бы оказаться более удобным способом размещения таких макетов.

Примером может служить столбчатая диаграмма в нашей информационной панели. В макете для компьютера названия фреймворков отображаются вдоль оси x, а процент осведомленности – вдоль оси y. Взглянув на эту диаграмму на мобильном устройстве, вы увидите, что названия фреймворков сжаты и их трудно читать.

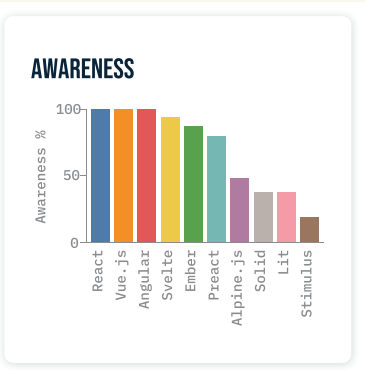
Упражнение: изменение ориентации столбчатой диаграммы на мобильном устройстве

Измените ориентацию столбчатой диаграммы при выборе мобильного макета, сделав столбцы горизонтальными и отобразив названия фреймворков вдоль оси y. Вот некоторые рекомендации, которые помогут вам в этом.

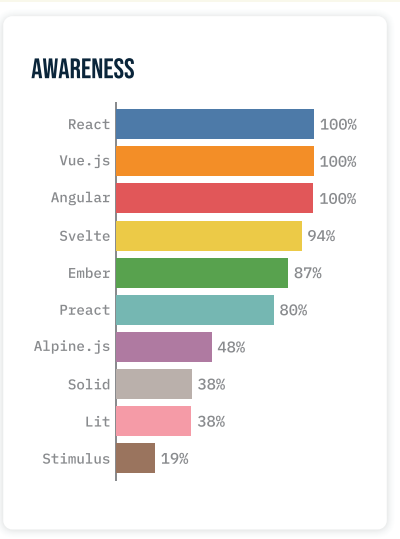
1. Создайте новый компонент с именем `BarChartMobile`, который будет отображать гистограмму в мобильном макете.

2 В `Charts.js` если переменная `layout` имеет значение `"desktop"`, то вызовите компонент `BarChart`, в противном случае – `BarChartMobile`.
3 Чтобы отобразить имена фреймворков вдоль оси y, создайте в `Axis.js` новую функцию генератора вертикальной оси для интервальной шкалы.
4 Вместо отображения оси с процентами осведомленности покажите округленные значения в конце каждого столбика, как показано на рисунке ниже.

Макет для компьютера и планшета



Макет для мобильного устройства



Изменение ориентации диаграммы – удобная стратегия для адаптивных визуализаций. В нашем примере при отображении на экране мобильного устройства метки с названиями фреймворков могут сжиматься и становиться нечитаемыми. Мы можем выделить для них больше места, отобразив вдоль вертикальной оси. Также проценты осведомленности можно показать в виде меток, а не на оси

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в листинге D.15 и в папке `9.3-Responsive_dashboard/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

9.4 Дополнительные советы

В этой главе было представлено множество конкретных советов по переносу цифровых визуализаций на небольшие экраны. Но это да-

леко не исчерпывающее обсуждение. Вот еще несколько идей, которые помогут вам адаптировать свои проекты.

- Некоторые типы визуализаций, такие как сложные радиальные диаграммы, плохо адаптируются к мобильным экранам. В таких случаях для отображения на мобильных устройствах, возможно, стоит выбрать совершенно другой тип диаграммы.
- Некоторые проекты, такие как дашборды бизнес-аналитики (Business Intelligent, BI), могут вообще быть недоступными на телефонах. В таких случаях достаточно определить только макет для компьютера.
- Сложные визуализации, содержащие сотни и тысячи элементов, могут потребовать значительных ресурсов, что особенно важно для мобильных устройств. Нестабильное подключение тоже может быть проблемой при загрузке больших наборов данных на мобильные устройства. Если визуализация не будет меняться со временем и не является интерактивной, то имеет смысл для мобильных макетов формировать статические изображения.
- Если это уместно, то добавьте возможность масштабирования и прокрутки визуализаций в мобильных макетах. Это особенно актуально для карт и сетевых диаграмм.
- Сделайте основную информацию легкодоступной. В мобильных макетах взаимодействие с визуализациями может быть обременительным, а подсказки работают не очень хорошо, поэтому весьма желательно предусмотреть отображение всей важной информации непосредственно на экране.
- Пальцы намного больше указателя мыши! Если вы планируете сделать свой проект интерактивным на мобильных устройствах, то подумайте о том, как сделать удобнее запуск сенсорных событий.
- Для визуализаций, которые занимают много места по горизонтали и при этом юриентацию которых нежелательно менять, можно рассмотреть возможность разрешить прокручивать их горизонтально. Такой подход позволяет размещать широкие диаграммы на ограниченном пространстве и упрощает их изучение.

Вот ресурсы, которые помогут вам глубже изучить адаптивную визуализацию:

- статья «Techniques for Data Visualization on Both Mobile & Desktop» Надии Бремер (Nadieh Bremer), http://mng.bz/PZA5;
- статья «Designing for Small Screens: Responsive Data Viz, Resizing, and Aspect Ratios» Дианы Макдональд (Diana MacDonald), http://mng.bz/7dX4;
- книга «Building Responsive Data Visualization for the Web», автор Билл Хиндерман (Bill Hinderman) (Wiley, 2015).

Интервью с Михаем Минко

Минко – эксперт по визуализации данных.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

После изучения философии в Сегедском университете (Венгрия) я заинтересовался инфографикой и визуализацией данных. Я начал работать над проектом дашборда для финансового директора университета, что, по сути, и положило начало моей карьере в области визуализации данных. Мне приходилось проектировать дашборды с использованием разных инструментов, таких как Processing и D3.js. Это было более 10 лет назад, и они до сих пор мои любимые инструменты. В настоящее время я работаю исследователем в Data Storytelling Hub в художественном университете имени Ласло Мохой Надь (Венгрия), где мы исследуем различные вопросы, связанные с визуализацией, а также занимаемся проектной работой для компаний и государственных организаций.

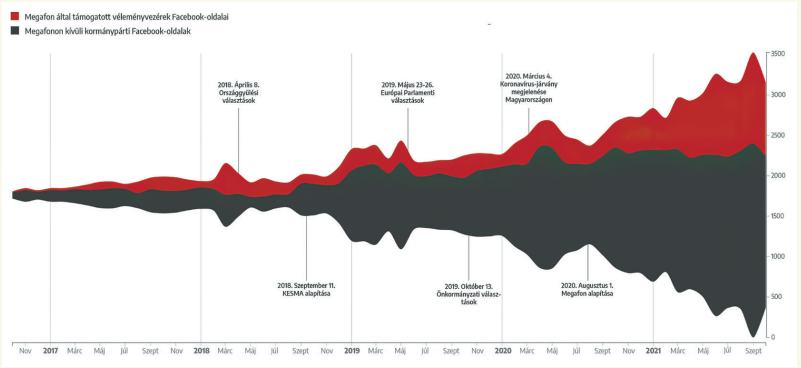
Какие инструменты визуализации данных вы используете постоянно? Есть ли среди них любимые?

В основном я использую инструменты разработки дашбордов, чаще всего D3.js, Processing и Gephi. В последнее время я начал использовать Blender, потому что он позволяет быстро создавать модели, учитывающие физические характеристики, и добавлять свет и текстуры. Когда я работаю один, то предпочитаю Tableau за то, что он позволяет быстро подключаться к широкому спектру источников данных и создавать прототипы дизайна. Когда я работаю в команде, то в основном использую D3.js, потому что эта библиотека очень легко настраивается и встраивается в различные системы публикации. D3.js также позволяет добиться высокого уровня интерактивности, который не могут обеспечить другие инструменты.

Какую роль играет D3.js и какое место она занимает в вашем рабочем процессе?

D3.js играет важную роль в моей работе, потому что это единственный инструмент, который в полной мере соответствует моим потребностям, когда требуется развернуть визуализацию в онлайн-СМИ. Чаще всего я работаю в сотрудничестве с другими, и эта библиотека позволяет нам быстро создавать прототипы и доводить их до окончательной версии, оставаясь в тесном взаимодействии. Мы можем работать вместе над разными слоями визуализации и тут же видеть результат. Кроме того, это самый универсальный инструмент из тех, что я знаю, позволяющий быстро опробовать новые типы визуализаций. Я часто использую ее в качестве промежуточного шага, когда публикую что-то в печатных СМИ, поскольку могу экспортировать сложные визуализации в программное обеспечение для настольных издательских систем.

Представленный проект: «The Metaverse of the Hungarian National Cooperation System» (https://atlo.team/ner-metaverzum/)



Потоковый график, визуализирующий количество страниц лидеров мнений в Facebook, поддерживаемых программой Megafon (красная область), и количество проправительственных страниц в Facebook, не относящихся к Megafon (черная область)

Опишите свой проект «NER Metaverzum», опубликованный на atlo.team (венгерский сайт, специализирующийся на журналистике данных) и созданный в сотрудничестве с Эстер Катона (Eszter Katona) и Сильви Немет (Szilvi Német).

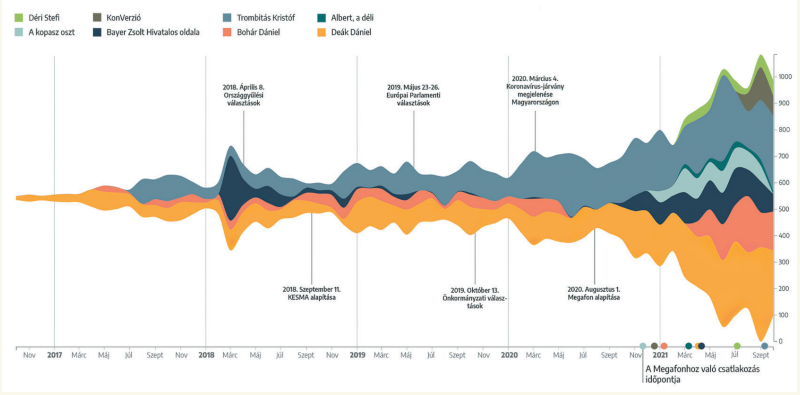
В 2010 году Facebook начал набирать популярность в Венгрии. Это был год, когда нынешняя правящая партия FIDESZ (Венгерский гражданский альянс) выиграла парламентские выборы, с тех пор находится у власти. Однако их признание важности социальных сетей запаздывало до 2020 года. Вот почему мы выбрали его в качестве объекта нашего исследования: кто составляет проправительственную информационную экосистему в социальной сети в определенный период в 2021 году. Понимая важность социальных сетей, было создано несколько десятков новых проправительственных страниц в Facebook. Правительство косвенно поддерживает своих влиятельных лиц. Также была запущена обучающая программа под названием «Megafon » для распространения проправительственных сообщений. Как выяснилось, создание «Megafon» оказало огромное влияние на вселенную венгерских социальных сетей. Это, возможно, также помогло FIDESZ победить на выборах 2022 года.

Мы использовали методы сетевого анализа и визуализации, чтобы раскрыть возможные связи между основными игроками в этой области. Мы измерили, как изменилось количество опубликованных сообщений СМИ, связанных с Megafon, и на основе этого анализа выявили наиболее важных игроков. Нам удалось собрать их сообщения в Facebook до того, как они стали частью Megafon, и, как показали полученные результаты, количество их сообщений значительно возросло после присоединения к Megafon.

Расскажите, как создавался этот проект.

Мы использовали CrowdTangle для получения сообщений, опубликованных выбранными влиятельными лицами. Как показал наш опыт,

в подобных проектах важно заранее определить объем данных, которые нужно получить, потому что без этого на последующих этапах их обработка может оказаться непосильной. После получения данных мы применили некоторые виды предварительной обработки, чтобы получить стандартизированный набор данных. Затем, недолго посоветовавшись, решили использовать метафору потокового графика, потому что хотели показать динамику изменения количества сообщений, а потоковый график подходит для этого лучше всего. Мы использовали D3.js, и, что удивительно, форма графика напомнила нам мегафон (рупор). Сама реализация графика не вызвала никаких сложностей, потому что в интернете можно найти сотни хороших примеров, на которых можно учиться. Сначала мы думали создать статическую визуализацию, чтобы поддерживаемые взаимодействия не отвлекали пользователя. Мы также хотели дать посетителям возможность загружать и использовать изображения как отдельные картинки. Вот почему на последнем этапе мы импортировали файл SVG из D3.js в Adobe Illustrator, чтобы добавить в него некоторые мелкие детали, а также чтобы опубликовать его в печатном издании *Átlátszó*, название которого переводится как «прозрачный».



Количество сообщений на страницах проправительственных влиятельных лиц в Facebook

Вы разработали потоковый график с помощью D3.js, а затем экспортировали его в Adobe Illustrator, чтобы улучшить визуальные эффекты и добавить аннотации. Часто ли вы используете такой гибридный подход в своей работе? В чем его преимущество?

Да, я нередко использую такой подход. Он дает возможность контролировать каждый шаг в процессе, а также подготовить окончательную визуализацию для печати. Я довольно часто использую разные системы или фреймворки вместе, потому что так я могу объединить сильные стороны этих фреймворков. Иногда я даже создаю физические объекты и для этого использую лазерную гравировку или резку, что требует другого типа подготовки. Для тех, кто знает, как эти инструменты могут работать вместе, открывается совершенно новый мир.

Вы преподаете визуализацию данных. Какими советами вы могли бы поделиться с теми, кто делает первые шаги в этой области? Есть ли какие-то особые препятствия, с которыми сталкиваются ваши ученики?

Мой первый совет, которому я, к сожалению, сам не следовал в свое время: рисуйте. Делайте наброски. Рисуйте часто и много. Рисование карандашом (или красками) на бумаге помогает развить творческое мышление. Однако для этого требуется одно важное условие: знать, как преобразовать данные в визуальные атрибуты на бумаге. Это может стать препятствием, но с терпением и практикой связь между ними обязательно появится. Воплотить рисунки и наброски в цифровые визуализации можно с помощью библиотеки D3.js, которая, по моему мнению, является наиболее удобной средой для создания визуализаций данных.

Итоги

- В настоящее время бóльшая часть интернет-контента потребляется на телефонах. Поэтому адаптация визуализаций для небольших экранов имеет решающее значение для охвата большей аудитории.
- Методы создания адаптивных визуализаций данных не так хорошо известны, как методы разработки адаптивных веб-сайтов. Стратегии зависят от типов диаграмм и аудитории.
- В зависимости от приоритетов и предпочтений мы можем создавать адаптивные дизайны двумя способами: с упором на мобильные устройства или с упором на настольные компьютеры.
- Не существует универсального способа создания адаптивных визуализаций. В этой главе мы использовали линейную шкалу для динамической адаптации размера меток в зависимости от ширины экрана. Мы также сократили количество рисок на оси и удалили вторичную информацию.
- Адаптивные сетки облегчают разработку адаптивных макетов. Такие макеты могут включать несколько визуализаций или смесь визуализаций и текста.
- Для диаграмм, занимающих много места по горизонтали, можно рассмотреть возможность изменения их ориентации на телефонах (при условии что телефон используется в портретном режиме).

10 Доступность визуализаций для людей с ограниченными возможностями

В этой главе рассматриваются:

- особенности потребления веб-контента людьми с ограниченными возможностями;
- оформление текста и инструкций с целью упростить их чтение;
- обеспечение цветового контраста и использование двойного кодирования;
- упрощение навигации для пользователей программ чтения с экрана;
- организация взаимодействий с помощью клавиатуры.

При проектировании и разработке визуализаций данных мы склонны сосредоточиваться на эстетической стороне: выборе цветовой палитры и шрифта, которые соответствуют определенному внешнему виду или разработке вызывающих восхищение нестандартных взаимодействий и анимаций. Но мы мало уделяем внимания доступности визуализаций для людей с ограниченными возможностями.

В США 26 % взрослых имеют инвалидность, а 4,9 % имеют нарушения зрения, требующие использования программ чтения с экрана (см. статью «57 Web Accessibiity Statistics» Джеффа Кадда (Geoff Cudd), https://ddiy.co/web-accessibility-statistics). Но в 2022 году толь-

ко 3 % интернета были доступны для людей с ограниченными возможностями (см. статью «What's Next for Digital Accessibility» Дэвида Моради (David Moradi), http://mng.bz/X10M).

В сфере визуализации данных мы сталкиваемся с дополнительными проблемами, обусловленными тем, что визуализации по своей природе очень наглядны, а структура наших проектов менее предсказуема, чем у традиционных веб-сайтов. Однако самое интересное, что для людей с ограниченными возможности нам нужно создавать не идентичный, а скорее эквивалентный опыт. Это означает, что мы должны дать им доступ к основным выводам, которые люди без инвалидности могут сделать сами, просматривая наши проекты и взаимодействуя с ними. Это также может означать предоставление упрощенного доступа к базовым данным.

Мы часто слышим, что чтобы сделать наши проекты доступными, требуется дополнительное время, бюджет и знания. Но, уделяя внимание некоторым ключевым аспектам на этапе разработки, мы можем значительно улучшить доступность без дополнительных затрат. На самом деле гораздо проще и дешевле учитывать доступность при создании проекта, чем пытаться сделать проект доступным, когда он уже создан. И стратегии, используемые для повышения доступности, часто улучшают проект, делая его более полезным для всех. В этой главе мы покажем, как это сделать!

10.1 Как люди с ограниченными возможностями потребляют веб-контент

Люди с ограниченными возможностями просматривают веб-страницы, используя адаптивные стратегии и вспомогательные технологии. К адаптивным стратегиям относятся увеличение размера текста, масштабирование изображений, снижение скорости движения указателя мыши, доступ к контенту с помощью клавиатуры и включение субтитров к видео.

К вспомогательным технологиям относятся программы чтения с экрана (экранные дикторы), экранные лупы, программное обеспечение для распознавания речи и переключатели управления. *Экранные дикторы* выводят веб-контент в виде речи или шрифта Брайля. Они в основном используются слепыми или людьми с очень ограниченным зрением. С другой стороны, люди с ослабленным зрением, как правило, используют экранные лупы – устройства или программы, увеличивающие информацию, отображаемую на экране. Такие инструменты также могут улучшать цветовой контраст, увеличивать размер значков или указателя мыши. Программное обеспечение для

распознавания речи может перемещаться по веб-странице, активировать ссылки и кнопки или заполнять текстовые поля голосовыми командами. Программное обеспечение этого типа полезно для людей, которые не могут использовать клавиатуру или мышь из-за травм, физических недостатков, хронических заболеваний или даже когнитивных нарушений. Недавно, с появлением голосовых помощников, таких как Алиса или Маруся, они также стали пользоваться популярностью среди людей без инвалидности. Наконец, переключатели управления выглядят как одна или несколько больших кнопок, которые отправляют сигнал нажатия клавиши на устройство для навигации в интернете.

10.2 Соответствие стандартам доступности

В рамках инициативы по обеспечению доступности веб-сайтов (Web Accessibility Initiative, WAI) консорциума World Wide Web Consortium (W3C) были разработаны руководящие принципы доступности под названием «Руководящие принципы доступности веб-контента» (Web Content Accessibility Guidelines, WCAG). Они побуждают нас создавать воспринимаемые, доступные, понятные и надежные проекты. Под *простотой восприятия* подразумевается наличие текстовой альтернативы для нетекстового контента, такого как изображения и видео. Подобранные цвета обеспечивают достаточный контраст, и пользователю даются достаточные подсказки, когда контент находится под указателем мыши или в фокусе ввода. *Доступные* проекты гарантируют возможность навигации по контенту с помощью клавиатуры, наличие у пользователя достаточного времени для чтения и использования контента или достаточную величину кнопок, чтобы на них могли нацеливаться люди с ограниченными двигательными навыками. Чтобы сделать наш контент *понятным*, мы подбираем текстовое содержимое так, чтобы оно могло быть понято большинством людей, веб-функции работают предсказуемым образом, а формы помогают пользователям избегать ошибок и исправлять их при необходимости. Наконец, мы делаем наши проекты *надежными*, гарантируя возможность их интерпретации различными пользовательскими агентами, такими как браузеры и вспомогательные технологии. Мы привели лишь несколько примеров, а за более полной информацией рекомендуем обратиться к справочнику WCAG (www.w3.org/WAI/WCAG22/quickref/).

В этой главе мы используем список критериев Chartability (https://chartability.fizz.studio/), и, опираясь на него, улучшим доступность диаграммы рассеяния китообразных, построенной в главе 7 и показанной на рис. 10.1. Chartability – это набор тестов, помогающих обеспечить доступность визуализации данных. Всего в списке 50 тестов,

14 из которых считаются критическими важными. В этой главе мы сосредоточимся на этих 14 критических тестах. Но по мере того, как вы будете осваивать принципы доступности, мы советуем вам не останавливаться на достигнутом и продолжать пополнять свой список пройденных тестов, пока не выполните их все. Эта тема заслуживает отдельной книги, поэтому мы не можем претендовать на полноту охвата, уделив ей всего лишь одну главу. Тем не менее, уделяя внимание критическим тестам, предложенным Chartability, мы можем значительно улучшить опыт пользователей с ограниченными возможностями.

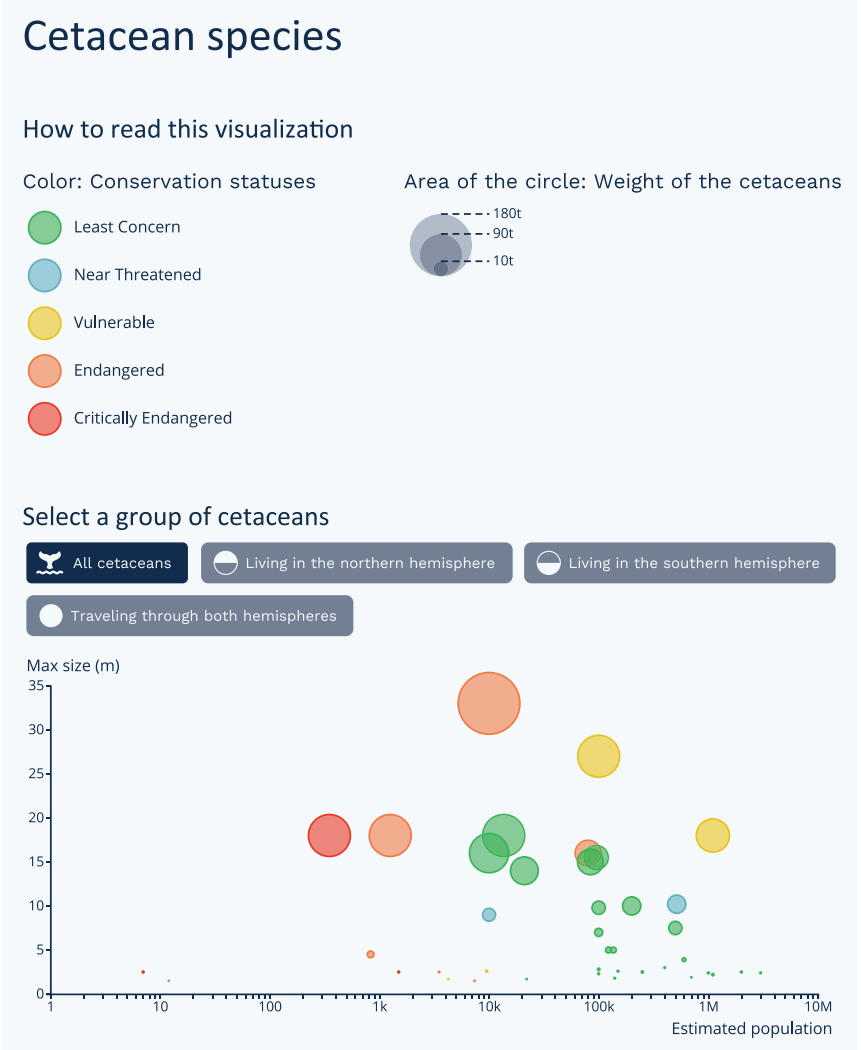


Рис. 10.1 Диаграмма рассеяния численности, размера и веса китообразных, построенная в главе 7. В этой главе мы улучшим ее доступность

В табл. 10.1 перечислены тесты, которые мы выполним в данной главе. Мы разбили этот список на пять разделов: текстовая информация, визуальная информация, доступность для экранного диктора, взаимодействия и прочее. Готовый проект вы найдете по адресу http://mng.bz/maB0.

Таблица 10.1 Контрольный список для проектов на основе D3.js

Текстовая информация
Объясняйте цель диаграммы, как ее читать, использовать и интерпретировать. Весь текст должен иметь уровень сложности 9 или ниже. Размер текста должен быть не менее 12 пикселей, а в идеале больше
Визуальная информация
Цвета должны быть достаточно контрастными. Информация должна передаваться не только с помощью цвета. Данные должны представляться с соответствующей плотностью
Доступность для экранных дикторов
Разметка должна быть структурирована семантически. Контейнер SVG должен предоставлять дополнительный контекст. Визуализированные данные должны быть доступны для чтения экранному диктору. В качестве альтернативы должна предоставляться таблица данных. Пользовательские настройки клавиатуры и сенсорного управления не должны совпадать с комбинациями управления экранным диктором
Взаимодействия
Пользователи должны иметь возможность осуществлять взаимодействия с помощью клавиатуры. Навигация и взаимодействия должны быть простыми и не требовать больших трудозатрат. Предусмотреть подсказки, помогающие осуществлять взаимодействия
Прочее
Проект должен обеспечивать эквивалентный опыт на экранах всех размеров. Изменения стиля, внесенные пользователем, не должны отменяться. Визуальные эффекты не должны представлять опасности для лиц, страдающих эпилептическими припадками

10.2.1 Текстовая информация

Визуализации данных редко располагаются на странице в гордом одиночестве. Обычно они сопровождаются текстом, содержащим описание и инструкции. В этом разделе мы поговорим о том, какую информацию следует добавлять в проекты для улучшения их доступности, как упростить чтение текста и как правильно его форматировать.

Объясняйте цель диаграммы, как ее читать, использовать и интерпретировать

Первый принцип доступности в нашем списке: объясняйте цель диаграммы, как ее читать, использовать и интерпретировать. Во вводном разделе проекта (рис. 10.2) мы объясняем суть визуализации, что означают круги, какой смысл вложен в их местоположение, размеры и цвет. Кроме того, визуализация имеет легенду, дополнительно описывающую цвета и размеры кругов.

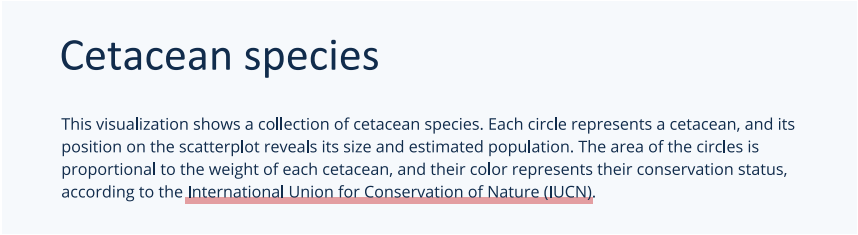


Рис. 10.2 Вводный раздел объясняет суть диаграммы и как ее читать

Метка **Select a Group of Cetaceans** (Выберите группу китообразных) сообщает, как использовать кнопки над визуализацией, а оси имеют подписи. Так что мы уже неплохо помогли пользователям интерпретировать наш проект.

Весь текст должен иметь уровень сложности 9 или ниже

Мы также должны гарантировать, что весь текст в нашем проекте рассчитан на читателя с уровнем образования 9 классов или ниже. Некоторые проекты требуют сложной терминологии, и для таких терминов мы должны предусмотреть простые определения. Чтобы оценить, на какой уровень образования рассчитан тест, Chartability предлагает использовать приложение Hemingway (https://hemingwayapp.com/). На рис. 10.3 можно видеть, как мы ввели основные фрагменты текста из проекта в интерфейс приложения. Приложение Hemingway сообщило нам, что в целом текст имеет уровень сложности 9 классов, что очень хорошо. Но оно также указывает, что введение содержит одно трудночитаемое предложение и одно очень трудно читаемое. На самом деле если ввести в приложение только текст введения, то уровень сложности поднимется до 14.

Чтобы улучшить читаемость текста, упростим введение. Предложение «Каждый круг представляет вид китообразных, а его местоположение на диаграмме рассеяния соответствует максимальному размеру особей и численности популяции» считается трудным для чтения, и его можно разбить на два предложения. Аналогично поступим со вторым проблемным предложением: «Площадь кругов пропорциональна максимальному весу особей каждого вида китообразных, а их цвет представляет природоохранный статус по данным Международного союза охраны природы (МСОП)». На рис. 10.4 видно, что мы переместили ссылку на МСОП в раздел «Методология», чтобы еще больше упростить текст. Теперь введение стало намного проще и понятнее.

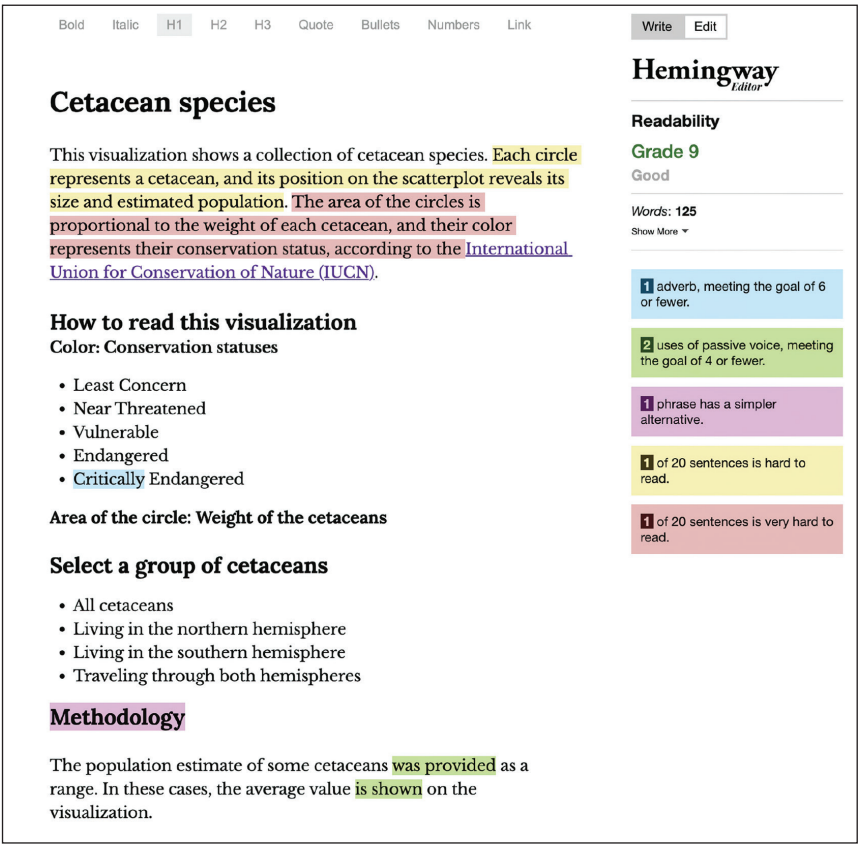


Рис. 10.3 Согласно приложению Hemingway, тексты нашего проекта соответствуют уровню сложности 9, что нам и нужно. Но оно также указывает, что одно предложение во введении трудно читать, а другое очень трудно читать (https://hemingwayapp.com/)

По предложению Heminway мы также заменили подзаголовок «Методология» более простой альтернативой «Методика». В целом изменения снизили уровень сложности чтения до 8. Обратите внимание, что такие корректировки также часто зависят от вашей аудитории и стиля, принятого в вашей организации. Но помните, что ясность всегда предпочтительнее литературного стиля.

В файле `index.js`, который находится в папке `10.2.1-Textual_information/start` в репозитории этой главы, мы обновили текст соответствующим образом. Окончательный текст можно увидеть в онлайн-версии проекта по адресу http://mng.bz/maB0.

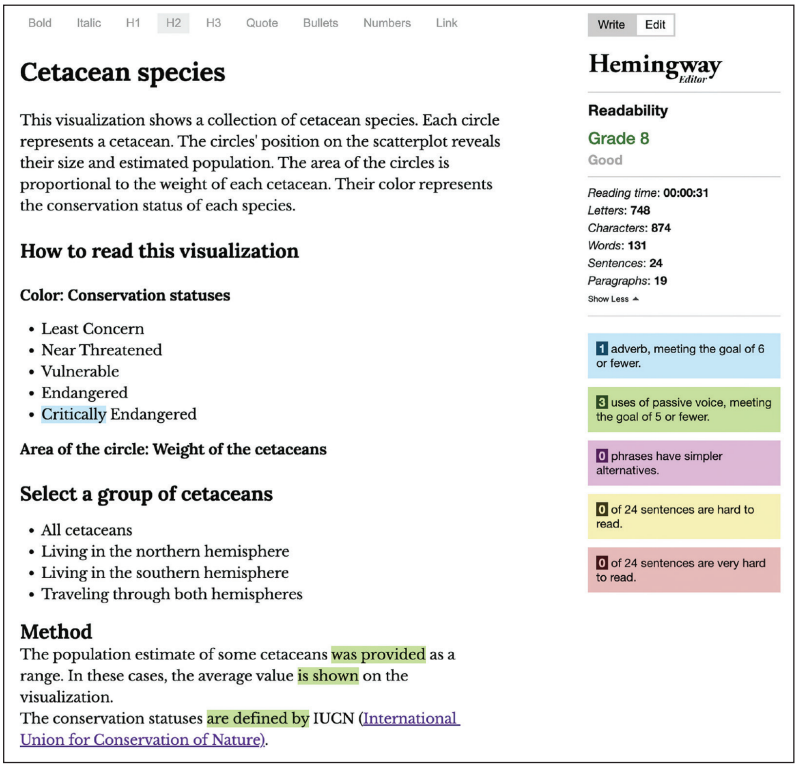


Рис. 10.4 После нескольких незначительных изменений уровень сложности чтения текста снизился до 8

Размер текста должен быть не менее 12 пикселей, а в идеале больше

Для оформления текста в Chartability рекомендуется использовать шрифт с размером не менее 12 пикселей. В этом списке правил также уточняется, что таким мелким шрифтом можно оформлять только вторичный текст, не оказывающий существенного влияния на понимание сути проекта, а бóльшая часть текста должна набираться более крупным шрифтом. В версии проекта для компьютеров основной текст желательно набирать шрифтом с размером 16 пикселей, а метки осей – 15 пикселей. Но если размер экрана уменьшится, то все элементы SVG станут меньше, включая текст.

Чтобы убедиться, что метки осей достаточно велики на маленьких экранах, можно заранее рассчитать фактический размер шрифта, поскольку он пропорционален ширине контейнера SVG. Мы знаем, что когда контейнер SVG имеет ширину 1200 пикселей, метки осей отображаются с размером 15 пикселей. На iPhone SE с шириной экрана 375 пикселей ширина контейнера SVG составит 315 пикселей (мы резервируем по 30 пикселей отступа с каждой стороны).

Простой способ масштабирования – использовать математическое правило пропорций. Согласно этому правилу, если число A пропорционально числу B, то мы можем найти X, которое пропорционально числу C, умножив B на C и разделив на A:

$$A \sim B,$$
$$C \sim X,$$
$$X = B \times C \div A.$$

Как показано на рис. 10.5, если размер текста составляет 15 пикселей, а ширина контейнера SVG 1200 пикселей, то согласно правилу пропорций при ширине контейнера SVG в 315 пикселей размер текста будет равен 4 пикселям, что очень мало:

$$315 \times 15 \div 1{,}200 \approx 4.$$

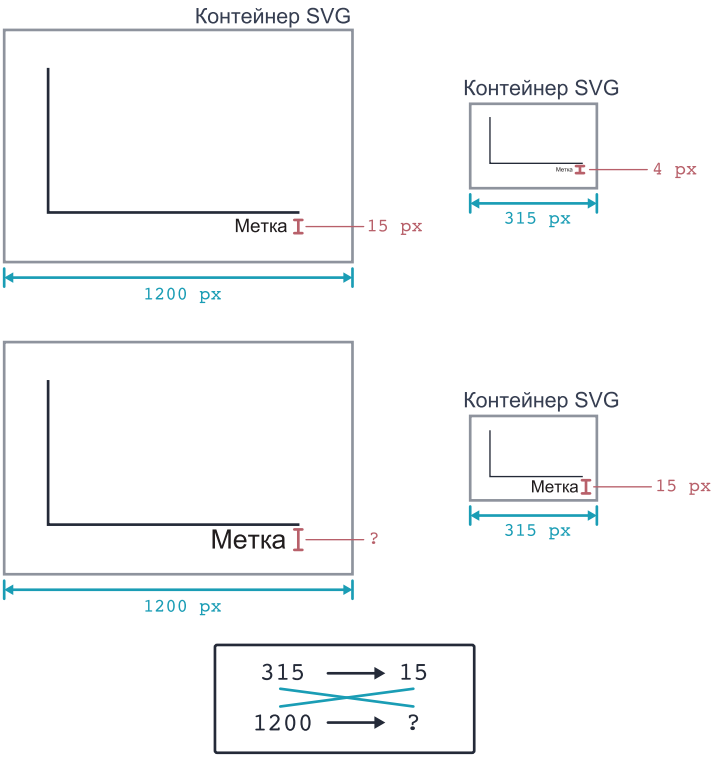


Рис. 10.5 Поскольку контейнер SVG имеет настроенный атрибут viewBox, но не имеет атрибутов width и height, все графические элементы будут изменять свои размеры вместе с экраном. Если размер шрифта меток осей задан равным 15 пикселям, то эти метки будут выглядеть намного меньше на экранах телефонов. Мы можем использовать правило пропорций, чтобы вычислить фактический размер шрифта, который нужно использовать на экранах телефонов

Чтобы на экране телефона метки выглядели так же, как метки высотой 15 пикселей на экране компьютера, их фактический размер по отношению к контейнеру SVG должен составлять около 57 пикселей:

$$15 \times 1{,}200 \div 315 \approx 57.$$

Как обсуждалось в главе 9, для динамического изменения размера текста в зависимости от ширины экрана можно использовать линейную шкалу. В `shared-constants.js` мы объявили линейную шкалу и назвали ее `fontSizeScale`. Предметная область связана с шириной контейнера SVG, которая варьируется от 315 пикселей на iPhone SE до 1200 пикселей на экранах настольных компьютеров, а диапазон определяется желаемым размером шрифта на экранах разной ширины. В макете для настольного компьютера размер шрифта должен составлять 15 пикселей, а на телефонах увеличиваться до 57 пикселей, согласно вычислениям выше. Мы также включили свойство `clamp()`, чтобы не позволить D3 экстраполировать значения за пределы диапазона:

```
const fontSizeScale = d3.scaleLinear()
  .domain([315, 1200])
  .range([57, 15])
  .clamp(true);
```

Затем в `resize.js` мы добавляем обработчик события изменения размеров окна и вызываем функцию `resizeChart()`, как показано в листинге 10.1. Функция извлекает свойство `innerWidth` объекта `window` и вызывает `fontSizeScale` для настройки стилевого свойства `font-size` меток, а также изменяет вертикальное положение меток, изменяя атрибут `y`.

Листинг 10.1 Изменение размера шрифта меток осей (resize.js)

```
const resizeChart = () => {

  windowWidth = window.innerWidth;

  d3.selectAll(".tick text, .axis-label")
    .style("font-size", `${fontSizeScale(
      windowWidth)}px`);
  d3.select(".axis-label-left")
    .attr("y", margin.top - fontSizeScale(windowWidth));
  d3.select(".axis-label-bottom")
    .attr("y", height - margin.bottom +
      2.2*fontSizeScale(windowWidth));

};
```

Получить значение свойства innerWidth объекта window и сохранить его в константе с именем windowWidth

Вызвать fontSizeScale, чтобы настроить стилевое свойство font-size меток осей

Скорректировать вертикальное местоположение меток осей

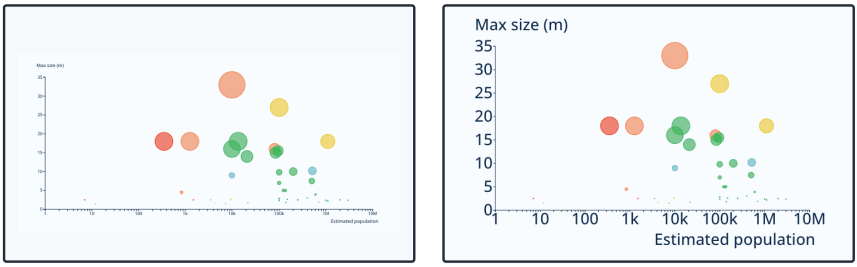
```
window.addEventListener("resize", () => {
  resizeChart();
});
```

Подключить обработчик события изменения размера окна, вызывающий функцию resizeChart()

Чтобы получить достаточно места для более крупных меток и подписей, мы также увеличиваем отступы и общую высоту:

```
const margin = {top: 110, right: 55, bottom: 130, left: 70};
const height = 800;
```

После внесения изменений сохраните проект и измените размер окна браузера. Вы увидите, что размер текста теперь достаточно велик для всех размеров экрана. Посмотрите, как выглядит онлайн-версия проекта (http://mng.bz/maB0), или взгляните на рис. 10.6: как видите, текст на диаграмме теперь достаточно велик и соответствует требованиям правил доступности.



До После

Рис. 10.6 Текст SVG действует подобно любому другому графическому элементу, поэтому метки осей выглядят крошечными на маленьких экранах. С помощью линейной шкалы текст может динамически изменять размер шрифта в зависимости от ширины экрана

Мы завершили приведение текстовой информации в соответствие с требованиями к доступности:

- объяснили назначение диаграммы, как ее читать, использовать и интерпретировать;
- весь текст имеет уровень сложности не выше 9;
- размер текста не менее 12 пикселей, а в идеале больше.

10.2.2 Визуальная информация

В этом разделе мы сосредоточимся на доступности визуальной информации в нашем проекте. Мы обеспечим достаточно высокую контрастность цветов, передачу информации не только цветом и организацию данных с соответствующей плотностью.

Цвета должны быть достаточно контрастными

В соответствии с руководящими принципами доступности веб-контента набор правил Chartability рекомендует, чтобы все геометрические фигуры и крупный текст имели контрастность по отношению к фону не менее 3:1, тогда как обычный текст должен иметь контрастность не менее 4,5:1. Измерить контрастность текстовых элементов можно с помощью расширения WCAG Color contrast checker для Chrome. Для оценки контрастности геометрических элементов, таких как круги на нашей диаграмме рассеяния и легенда для кругов, нужно вручную ввести их цвет в инструмент проверки контрастности WebAIM contrast checker (https://webaim.org/resources/contrastchecker/).

На рис. 10.7 показаны результаты проверки контрастности. Текст на странице и текст на активной кнопке имеют коэффициент контрастности 12,91:1, что значительно выше минимального значения 4,5:1. С другой стороны, текст, отображаемый на неактивных кноп-

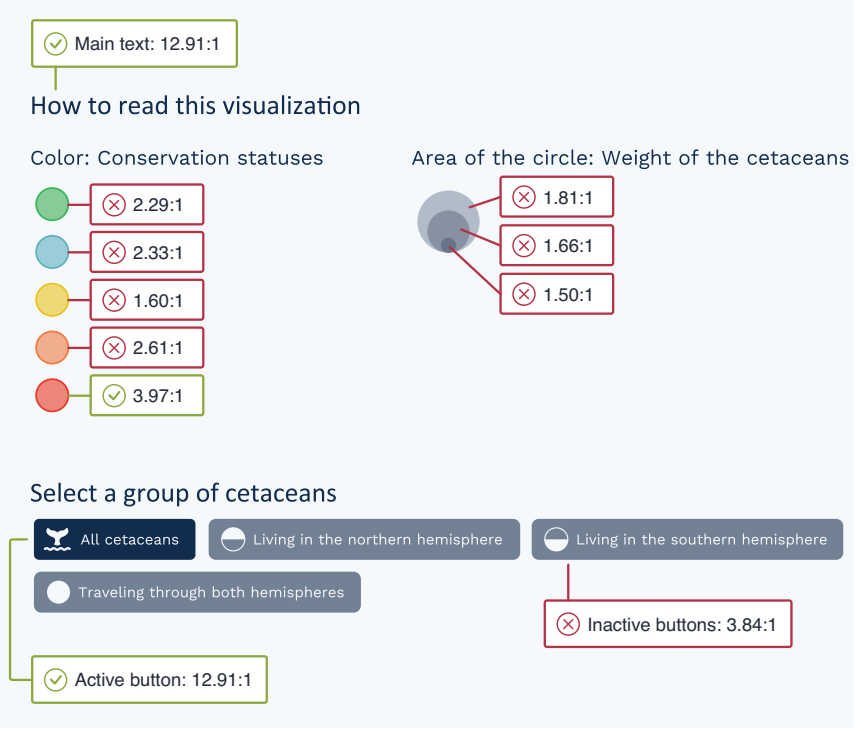


Рис. 10.7 Согласно расширению WCAG Color contrast checker, текст и активная кнопка на диаграмме имеют достаточный контраст. Но текст на неактивных кнопках недостаточно контрастен. Цвета, выбранные для представления природоохранных статусов и легенды, описывающей площади кругов, не создают достаточного контраста с фоном

ках, имеет коэффициент контрастности 3,84:1, т. е. он недостаточно контрастный. Для геометрических фигур, используемых в визуализации, коэффициент контрастности между цветом их границ и фоном должен составлять не менее 3:1. К сожалению, большинство цветов не соответствуют требованиям, как и геометрические фигуры, используемые в легенде площадей кругов.

Чтобы улучшить контрастность кругов, представляющих природоохранный статус, начнем с того, что изменим цвет фона на чисто белый (#ffffff). Далее, в инструменте оценки контрастности WebAIM есть ползунок для затемнения введенного цвета. Он динамически вычисляет новый коэффициент контрастности и сообщает, когда будет достигнут достаточный контраст. Но затемнение цветов может сделать их плоскими. Вместо этого можно поискать доступные цветовые палитры. На рис. 10.8 показана новая цветовая палитра, содержащая цвета, предложенные стандартом визуализации данных правительства США (http://mng.bz/5laq). Эти цвета обеспечивают более высокий контраст.

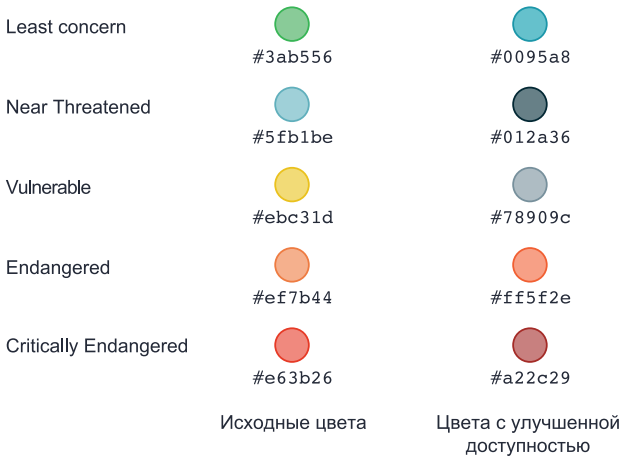


Рис. 10.8 Новые цвета, обозначающие природоохранный статус, обеспечивают достаточный контраст на белом фоне, что улучшает доступность

Теперь у нас есть цветовая палитра, обеспечивающая достаточный контраст. Новые контрасты показаны на рис. 10.9. Мы также увеличили непрозрачность фона неактивных кнопок с 60 % до 65 % и добавили темную обводку вокруг кругов в легенде. Добавление более темной обводки – полезная стратегия, когда нет возможности или желания менять цвет заливки.

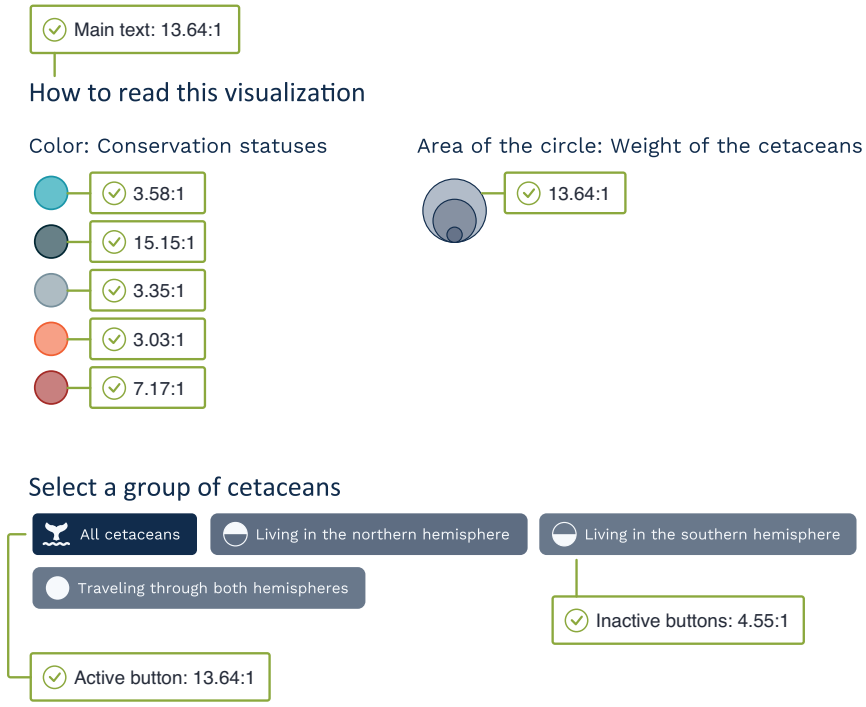


Рис. 10.9 Новые цвета обеспечивают достаточный контраст на белом фоне. Мы также добавили темную обводку вокруг кругов, используемых в легенде, и увеличили непрозрачность неактивных кнопок

Информация должна передаваться не только с помощью цвета

Теперь цвета достаточно контрастируют с фоном, но они могут оказаться трудноразличимыми для людей, страдающих дальтонизмом, которые видят цвета иначе, чем большинство из нас. Разные виды дальтонизма могут вызывать сложности с различением цветов или оттенков одного и того же цвета. Это также может повлиять на способность человека видеть, насколько яркими являются цвета. На рис. 10.10 показана наша палитра цветов, обозначающих природоохранный статус, как их видят люди, страдающие разными видами дальтонизма. Здесь можно использовать расширение WCAG Color contrast checker для Chrome.

Чтобы гарантировать, что люди с цветовой слепотой смогут различать природоохранные статусы, мы используем стратегию, называемую двойным кодированием. С помощью двойного кодирования мы объединим цветовую палитру с другим способом передачи информации. Например, мы могли бы придать разную форму каждому статусу или наложить разный узор на каждый цвет. Поскольку размер кругов на диаграмме рассеяния уже несет информацию (мак-

симальный вес особей китообразных), использование различных форм привнесло бы ненужную сложность. Вместо этого мы применим второй вариант: добавление узоров.

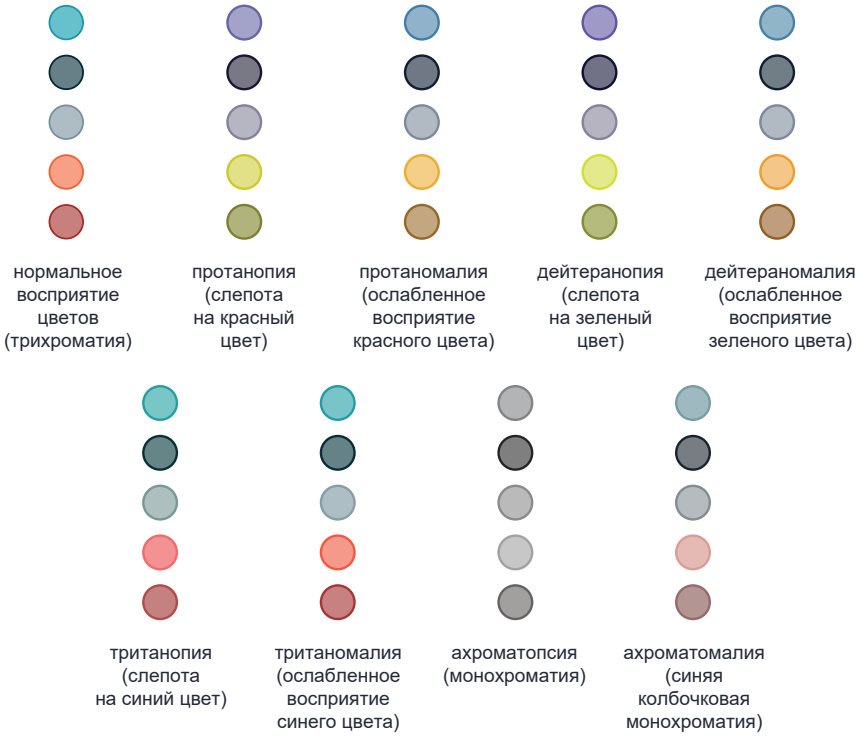


Рис. 10.10 Люди с разными формами дальтонизма по-разному воспринимают одни и те же цвета, что затрудняет различение цветов. В идеале мы должны передавать информацию не только с помощью цвета. Здесь показано, как люди с разными формами дальтонизма воспринимают цвета из нашей палитры

Чтобы создать узор SVG, сначала нужно добавить в контейнер SVG элемент `<defs>`. Этот элемент хранит графические объекты, которые позже будут использоваться в SVG. Он может содержать градиенты, размытия или, как в нашем случае, узоры. Добавим элемент `<pattern>` внутрь элемента `defs` и начнем создавать фигуры, составляющие узор.

Определим первый узор как череду горизонтальных линий. В конце функции `drawScatterplot()` в файле `scatterplot.js` добавим элемент `defs` в контейнер SVG. Далее, как показано в следующем фрагменте, добавим элемент `pattern` в `defs` и сохраним его в константе с именем `horizontalLinesPattern`. Присвоим ему идентификатор `"pattern-horizontal-lines"`. Позже мы будем использовать этот идентификатор для применения узора к желаемой фигуре, поэтому он должен быть уникальным. Зададим ширину и высоту узора

равными 4 пикселям и установим атрибут `patternUnits` в значение `userSpaceOnUse`, это означает, что геометрические свойства будут определяться относительно родительского элемента SVG:

```
const defs = d3.select("#scatterplot svg")
  .append("defs");

const horizontalLinesPattern = defs
  .append("pattern")
    .attr("id", "pattern-horizontal-lines")
    .attr("width", 4)
    .attr("height", 4)
    .attr("patternUnits", "userSpaceOnUse");
```

Фигуры SVG, определенные внутри элемента `pattern`, становятся плитками, повторяющимися в горизонтальном и вертикальном направлениях. Поэтому чтобы создать узор из чередующихся горизонтальных линий, можно определить плитку, состоящую из белого квадрата, на котором нарисована линия SVG, как показано на рис. 10.11.

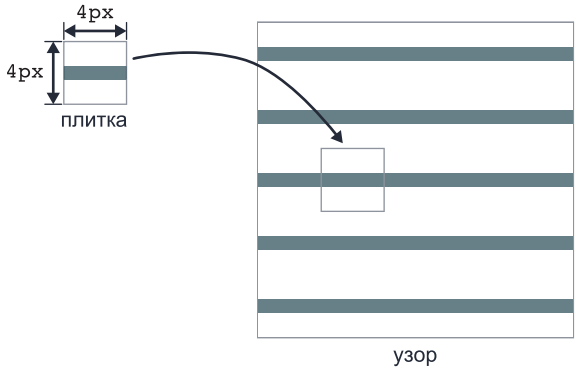


Рис. 10.11 Получить узор, состоящий из чередующихся горизонтальных линий, можно, создав плитку с белым квадратом и горизонтальной линией SVG. Эта плитка будет повторяться по горизонтали и вертикали, создавая желаемый узор

В следующем фрагменте мы сначала добавляем в узор элемент `rect` белого цвета. Устанавливаем его ширину и высоту равными 100 %, чтобы покрыть всю область плитки. Затем добавляем элемент горизонтальной линии в середину контейнера узора и в качестве цвета рамки назначаем цвет, связанный с природоохранным статусом Near Threatened (NT) – находящиеся в состоянии, близком к угрожаемому. Для этого вызываем цветовую шкалу, созданную в главе 7 и доступную в `scatterplot.js`:

```
horizontalLinesPattern
  .append("rect")
    .attr("x", 0)
    .attr("y", 0)
    .attr("width", "100%")
    .attr("height", "100%")
    .attr("fill", "#fff");
horizontalLinesPattern
  .append("line")
    .attr("x1", 0)
    .attr("y1", 2)
    .attr("x2", 4)
    .attr("y2", 2)
    .attr("stroke", colorScale("NT"));
```

Чтобы применить нужный узор, требуется возможность выбирать круги в диаграмме рассеяния по природоохранному статусу. Давайте поднимемся в начало `scatterplot.js` и найдем код, добавляющий круги в диаграмму. Каждому кругу мы придаем два имени класса: `"cetacean"` и `"cetacean-"`, за которыми следует природоохранный статус, указанный в данных, прикрепленных к каждому элементу. В нашем случае используются статусы: `LC` (Least Concern – вызывающие наименьшие опасения), `NT` (Near Threatened – находящиеся в состоянии, близком к угрожаемому), `VU` (Vulnerable – уязвимые), `EN` (Endangered – исчезающие) и `CE` (Critically Endangered – находящиеся под критической угрозой исчезновения):

```
innerChart
  .selectAll(".cetacean")
  .data(data)
  .join("circle")
    .attr("class", d => `cetacean cetacean-${d.status}`)
    ...
```

Теперь выберем все круги с классом `"cetacean-NT"` и заменим значение их атрибута `fill` на `url(#pattern-horizontal-lines)` – идентификатором, который мы определили для этого узора ранее:

```
d3.selectAll(".cetacean-NT")
  .attr("fill", "url(#pattern-horizontal-lines)");
```

После назначения узоров круги, соответствующие видам китообразных с природоохранным статусом Near Threatened (находящиеся в состоянии, близком к угрожаемому), будут выглядеть, как показано на рис. 10.12.

Другие узоры, которые мы создадим, – белый горошек для статуса Least Concern (вызывающие наименьшие опасения), и звездочки для статуса Endangered (исчезающий). Код в листинге 10.2 добавляет

в элемент `defs` новый контейнер узора. Присваивает этому контейнеру идентификатор `"pattern-circles"` и задает ширину и высоту, равные 5 пикселям. Как и ранее, атрибуту `patternUnits` присваивается значение `userSpaceOnUse`, чтобы геометрические свойства определялись относительно родительского контейнера SVG. И задает в атрибуте `patternTransform` смещение величиной в 2 пикселя с поворотом на 45°, чтобы создать более интересную компоновку.



Рис. 10.12 В соответствии с природоохранным статусом к кругам применяются соответствующие узоры. Сочетание цвета с узором является примером стратегии двойного кодирования

Далее в контейнер узора добавляется элемент `rect` и его атрибуту `fill` присваивается цвет, связанный с природоохранным статусом Least Concern (вызывающие наименьшие опасения). Наконец, в центр плитки добавляется белый круг с радиусом 1,5 пикселя.

Листинг 10.2 Создание узора белый горошек (scatterplot.js)

```
const circlesPattern = defs
  .append("pattern")
    .attr("id", "pattern-circles")
    .attr("width", 5)
    .attr("height", 5)
    .attr("patternUnits", "userSpaceOnUse")
    .attr("patternTransform", "translate(2)
      rotate(45)");
circlesPattern
  .append("rect")
    .attr("x", 0)
    .attr("y", 0)
    .attr("width", "100%")
    .attr("height", "100%")
    .attr("fill", colorScale("LC"));
circlesPattern
  .append("circle")
    .attr("cx", 2)
    .attr("cy", 2)
    .attr("r", 1.5)
    .attr("fill", "#fff");
```

Добавить новый узор в элемент defs. Установить его ширину и высоту равными 5 пикселям. Применить смещение с помощью атрибута patternTransform

Добавить элемент rect в узор и присвоить его атрибуту fill цвет, связанный с природоохранным статусом Least Concern

Добавить белый круг с радиусом 1,5 пикселя в центр плитки

Узор из звездочек для статуса Endangered (исчезающий) создается аналогично (см. листинг 10.3). Сначала мы добавляем новый узор в элемент `defs` и задаем ширину и высоту плитки равными 10 пикселям. Дополнительно применяем сдвиг на 2 пикселя и поворот на 45°.

Затем в узор добавляется белый прямоугольник, который будет служить фоном, и вслед за ним многоугольник в форме звезды. Элемент многоугольника – это замкнутая фигура, состоящая из прямых линий. Линии соединяются в точках с координатами, которые определены в атрибуте `points`. Атрибуту `fill` звезды мы присваиваем оранжевый цвет, соответствующий статусу Endangered (исчезающий).

Листинг 10.3 Создание узора из оранжевых звезд (scatterplot.js)

```
const starPattern = defs
  .append("pattern")
    .attr("id", "pattern-stars")
    .attr("width", 10)
    .attr("height", 10)
    .attr("patternUnits", "userSpaceOnUse")
    .attr("patternTransform", "translate(2)
      rotate(45)");
starPattern
  .append("rect")
    .attr("x", 0)
    .attr("y", 0)
    .attr("width", "100%")
    .attr("height", "100%")
    .attr("fill", "#fff");
starPattern
  .append("polygon")
    .attr("points", "0,0 2,5 0,10 5,8 10,10
      8,5 10,0 5,2")
    .attr("fill", colorScale("EN"));
```

Добавить новый узор в элемент defs. Установить его ширину и высоту равными 10 пикселям. Применить смещение и поворот с помощью атрибута `patternTransform`

Добавить элемент rect в узор

Добавить элемент polygon с атрибутом `points`, создающим звезду. Присвоить атрибуту `fill` цвет, связанный со статусом Endangered

Упражнение: создание шаблонов SVG

Теперь ваш черед! Создайте два новых узора SVG для природоохранных статусов Vulnerable (уязвимые) и Critically Endangered (находящиеся под критической угрозой исчезновения). Для обозначения уязвимых видов узор состоит из вертикальных серых линий на белом фоне. Для обозначения видов, находящихся под критической угрозой исчезновения, нарисуйте узор из белых крестов на красном фоне. Крест можно нарисовать с помощью двух линий или многоугольника. Получить цвет, связанный с тем или иным природоохранным статусом, можно, обратившись к шкале `colorScale()` и передав ей аббревиатуру статуса (`"VU"` для уязвимых видов и `"CR"` для находящихся под критической угрозой исчезновения).

При желании смело используйте другие узоры, которые вам больше нравятся! В поисках вдохновения можете также заглянуть на сайт https://pattern.monster/. Вы найдете решение в листинге D.16 и в папке `10.2.2-Visual_information/end`. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

После создания всех узоров SVG остается только применить их, присвоив атрибуту `fill` каждого круга в диаграмме рассеяния. В листинге 10.4 показана функция `getPattern()`, которая получает код статуса и возвращает соответствующий идентификатор узора. Затем внутри `drawScatterplot()` мы выбираем все элементы с классом `"cetacean"` и устанавливаем их атрибуты `fill`, вызывая `getPattern()` с соответствующим природоохранным статусом из данных, прикрепленных к каждому кругу.

Листинг 10.4 Применение узоров (scatterplot.js)

```
const drawScatterplot = (data) => {

  ...

  d3.selectAll(".cetacean")
    .attr("fill", d => getPattern(d.status))

};
```

Выбрать все элементы с именем класса "cetacean" и задать их атрибуты `fill`, вызвав функции getPattern() и передав ей статус из данных, прикрепленных к каждому кругу

```
const getPattern = (status) => {
  switch(status) {
    case "LC":
      return "url(#pattern-circles)";
    case "NT":
      return "url(#pattern-horizontal-lines)";
    case "VU":
      return "url(#pattern-vertical-lines)";
    case "EN":
      return "url(#pattern-stars)";
    case "CR":
      return "url(#pattern-cross)";
  };
};
```

Объявить функцию, которая получает статус и возвращает соответствующий идентификатор узора

После щелчка на той или иной кнопке фильтра над диаграммой рассеяния создаются новые круги и добавляются в визуализацию. В следующем фрагменте мы вызываем `getPattern()`, чтобы задать атрибут `fill` для каждого добавляемого круга (в `interactions.js`):

```
innerChart
  .selectAll("circle")
  .data(updatedData, d => d.uid)
  .join(
    enter => enter
      .append("circle")
        ...
        .attr("fill", d => getPattern(d.status))
        ...
    )
```

В `legend.js` поступаем аналогично с кругами, отображаемыми в легенде:

```
statuses
  .append("svg")
    .attr("width", 32)
    .attr("height", 32)
  .append("circle")
    ...
    .attr("fill", d => getPattern(d.id))
    ...
```

По завершении диаграмма рассеяния должна выглядеть так, как показано на рис. 10.13 или в онлайн-версии проекта (http://mng.bz/maB0).

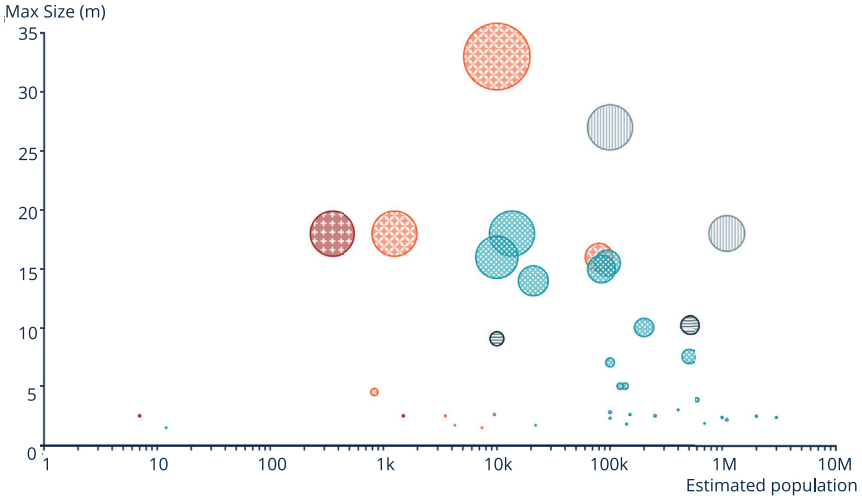


Рис. 10.13 Разные природоохранные статусы на диаграмме рассеяния мы обозначили не только разными цветами, но и разными узорами SVG. Это еще один пример стратегии двойного кодирования, которая помогает пользователям с цветовой слепотой различать категории

Данные должны представляться с соответствующей плотностью

На нашей диаграмме рассеяния круги не перекрываются, и это позволяет нам сказать, что данные представлены с соответствующей плотностью. В Chartability предлагается предоставлять дополнительные пояснения к скоплениям элементов визуализации, оказавшихся в одном месте на диаграмме, или агрегировать диаграмму до уровня с меньшим количеством элементов, как показано на рис. 10.14.

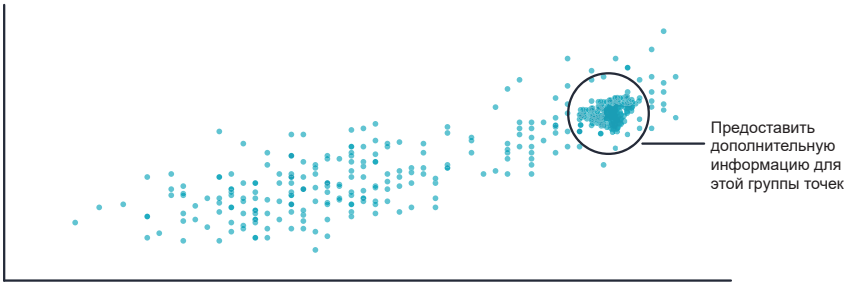


Рис. 10.14 В Chartability предлагается предоставлять дополнительные пояснения к скоплениям элементов визуализации, оказавшихся в одном месте на диаграмме, или агрегировать диаграмму до уровня с меньшим количеством элементов

Мы завершили приведение нашей визуализации в соответствие с рекомендациями по обеспечению доступности. Теперь:

- цвета имеют достаточный контраст;
- информация передается не только с помощью цвета;
- данные представлены с соответствующей плотностью.

10.2.3 Доступ к экранному диктору

Программы чтения с экрана (экранные дикторы) помогают слепым или людям с очень слабым зрением получать доступ к веб-контенту. Как следует из названия, такое программное обеспечение читает контент вслух. Чтобы пользователи экранного диктора имели доступ к информации, содержащейся в наших визуализациях, мы должны обратить внимание на несколько ключевых аспектов. Однако важно помнить, что цель заключается в предоставлении не идентичного, а *эквивалентного* опыта.

В качестве экранных дикторов можно использовать несколько различных программных приложений, включая Narrator на ПК и VoiceOver на Mac. В этой главе мы используем расширение ChromeVox для Chrome, поскольку его можно применять в любой операционной системе. Чтобы установить расширение, выполните следующие действия:

1. откройте Chrome;
2. перейдите по ссылке http://mng.bz/aMvX;
3. щелкните на кнопке **Add to Chrome** (Добавить в Chrome), чтобы установить расширение ChromeVox;
4. ChromeVox включится автоматически. Чтобы отключить его, перейдите в **Settings > Extensions** (Настройки > Расширения) и снимите флажок **Enabled** (Включено).

Когда открывается новая вкладка, расширение ChromeVox зачитывает вслух HTML-тег `<title>`, в нашем случае «Cetacean species | D3.js in Action». Вы можете найти этот текст в начале `index.html`, внутри тега `<head>`. Поскольку это первое, что слышат люди, использующие программы экранных дикторов, когда попадают на страницу, текст заголовка должен быть достаточно описательным.

Есть несколько способов навигации по странице с ChromeVox. Щелкая на элементах на странице, экранный диктор прочтет их вслух. Также можно использовать клавишу **Tab** для перехода к следующему элементу, например к кнопке или ссылке. Чтобы просмотреть все содержимое страницы, можно перемещаться с помощью **Control**+**Alt** в Windows и **Control**+**Cmd** в Mac, нажимая клавиши со стрелками вверх и вниз.

Используя экранного диктора для навигации по проекту, мы заметили, что ничего не сообщаем пользователям, когда они перемещаются по графическим элементам, таким как легенда или диаграмма рассеяния. Экранный диктор может дотянуться до чисел и меток осей, но они не имеют никакого смысла, если глаза не позволяют видеть диаграмму рассеяния. Кроме того, информация, которую несут круги на диаграмме рассеяния, вообще отсутствует. В этом разделе мы постараемся улучшить опыт пользователей экранных дикторов.

Разметка должна быть структурирована семантически

Чтобы программы экранных дикторов могли интерпретировать веб-страницу, важно гарантировать семантическое структурирование разметки, т. е. элементы HTML должны использоваться строго по назначению. Семантическую разметку HTML иногда называют старой доброй семантической разметкой HTML (Plain Old Semantic HTML, POSH). Правильное семантическое использование элементов полезно не только для пользователей программ экранных дикторов, они также улучшают поисковую оптимизацию (SEO)!

Ниже перечислено несколько важных аспектов, на которые следует обратить внимание.

- Используйте элементы заголовков (`<h1>`, `<h2>`, `<h3>` и т. д.) для оформления заголовков и подзаголовков. Веб-страница должна иметь только один элемент `<h1>`. Программы экранных дикторов уведомляют пользователей о наличии заголовка и могут

озвучивать список всех элементов заголовков для удобства навигации по оглавлению.

- Используйте семантические элементы, такие как `<header>`, `<nav>`, `<main>`, `<article>`, `<section>` и `<footer>`, для правильной организации макета страницы. В таком случае макет будет иметь логический смысл при чтении программой экранного диктора.
- Используйте семантические элементы для управления пользовательским интерфейсом, такие как кнопки (`<button>`), ссылки (`<a>`) и формы (`<form>`). Придавайте им осмысленные метки. Избегайте использования элементов `<div>` для управления пользовательским интерфейсом.
- Группируйте коллекции элементов в неупорядоченные (`<ul>`) или упорядоченные (`<ol>`) списки. При встрече со списком экранные дикторы уведомляют пользователя о количестве элементов в списке, помогая ему понять, насколько велика коллекция элементов.

В нашем проекте мы правильно используем элементы заголовков, и программа экранного диктора их упоминает. Но отсутствуют семантические элементы макета, такие как `<header>`, `<main>`, `<section>` и `<footer>`. Элемент `<header>` должен служить контейнером для вводного раздела. В листинге 10.5 мы заключаем заголовок `<h1>` и вводный абзац в элемент `<header>`. Затем открываем элемент `<main>` и перемещаем в него легенду и диаграмму рассеяния. Элемент `<main>` определяет основное содержимое страницы. Мы разбиваем это содержимое на два раздела: легенду и визуализацию, используя элементы `<section>`. Наконец, мы заключаем источники и методологию в элемент `<footer>`.

Листинг 10.5 Добавление семантических элементов макета (index.html)

```
<body>
  <div class="container">
    <header>
      <h1>Cetacean species</h1>
      <div class="intro">...</div>
    </header>
    <main>
      <section>
        <div class="legend-container">
          ...
        </div>
      </section>
      <section>
        <div class="filters-container">
          ...
        </div>
        <div class="responsive-svg-container">
          <div id="scatterplot"></div>
```

```
        </div>
      </section>
    </main>
    <footer>
      <div class="sources">
        ...
      </div>
      <div class="methodology">
        ...
      </div>
    </footer>
  </div>
  ...
</body>
```

В нашем проекте мы уже оформили элементы легенды и источники как неупорядоченные списки, что хорошо. Но то же самое мы должны сделать для кнопок, отображаемых над диаграммой, потому что они представляют собой отдельную коллекцию элементов. Такая организация поможет пользователям экранных дикторов узнать, что присутствует группа похожих элементов.

Кроме того, чтобы улучшить доступность кнопок, мы присваиваем их атрибуту `type` значение `button`, а атрибуту `aria-pressed` – значение `true`, если фильтр активен, и `false` в противном случае. Это позволит пользователям экранного диктора узнать, какая кнопка нажата в данный момент.

Листинг 10.6 Группировка фильтров (interactions.js)

```
const populateFilters = () => {

  const filters = d3.select("#filters")
    .append("ul")
    .selectAll(".filter")
    .data(cetaceanFilters)
    .join("li")
    .append("button")
      .attr("class", d => `filter filter-${d.id} ${d.isActive ?
      ➥ "active" : ""}`)
      .attr("type", "button")
      .attr("aria-pressed", d => d.isActive ? true : false);

  ...

};
```

Затем в `visualization.css` нам нужно немного изменить стили, применяемые к фильтрам:

```
#filters ul {
  display: flex;
  flex-wrap: wrap;
```

```
  margin: 0;
  padding-left: 0;
  list-style: none;
}
```

На рис. 10.15 показано, как семантические элементы распределены на странице.

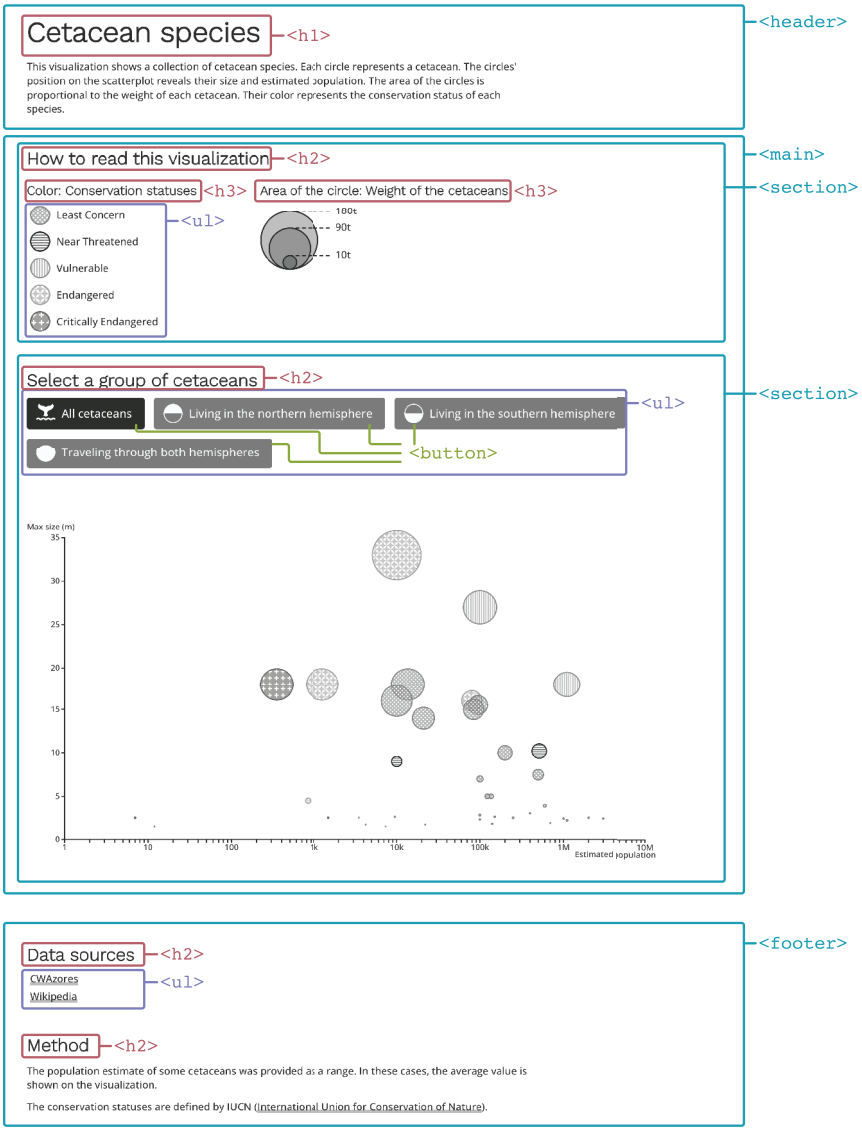


Рис. 10.15 Правильное использование семантических элементов HTML помогает пользователям программ экранных дикторов понять макет страницы и способствует оптимизации для поисковых систем (SEO)

Если вы снова откроете проект с помощью программы чтения с экрана, то заметите, что она упомянет элементы семантической разметки, такие как `<header>` и `<section>`, и правильно объявит набор кнопок.

Контейнер SVG должен предоставлять дополнительный контекст

Поскольку графика SVG похожа на изображения, экранным дикторам трудно ее интерпретировать. В настоящее время экранный диктор читает текст внутри диаграммы рассеяния SVG, например отметки осей, но не предоставляет дополнительной информации. Мы можем решить эту проблему, добавив теги `<title>` и `<desc>` в контейнер SVG, и они будут прочитаны вслух.

В листинге 10.7 мы сначала присваиваем значение `"img"` атрибуту `role` контейнера SVG. Этот атрибут определяет семантическое значение и помогает программам чтения с экрана узнать, как взаимодействовать с SVG. Атрибут `role` со значением `"img"` также запрещает экранному диктору читать текстовые метки внутри контейнера SVG. В зависимости от контекста это может быть вполне оправданно, потому что чтение отметок на осях не всегда полезно для пользователей. Но иногда бывает желательно позволить экранному диктору читать эти текстовые элементы, особенно на диаграммах, содержащих мало элементов.

Затем добавляем в контейнер SVG тег `title` с кратким названием графики SVG – в данном случае «Visualization of a collection of cetacean species» (визуализация коллекции видов китообразных) – и он должен быть первым дочерним элементом контейнера SVG. При наведении указателя мыши на SVG содержимое этого тега будет выводиться во всплывающей подсказке.

В теге `desc` мы предоставим информацию о целях этой диаграммы, основных выводах, которые можно сделать, глядя на нее, и основных тенденциях. Вот описание, которое мы будем использовать в нашей диаграмме рассеяния:

> *This scatterplot provides information about 39 cetacean species. The biggest cetacean is the blue whale, with a length of 33 meters and a weight of 173 tons. The Pantropical Spotted dolphin has the largest population, with 3 million individuals. The Baiji dolphin, the North Atlantic Right whale, and the Atlantic Humpback dolphin are in danger of extinction.*

Перевод:

> *Эта диаграмма рассеяния содержит информацию о 39 видах китообразных. Самый большой кит – синий кит, длина которого может достигать 33 метров, а вес – 173 тонн.*

Наибольшую численность популяции имеет пантропический пятнистый дельфин – 3 миллиона особей. Китайский озерный дельфин, североатлантический гладкий кит и атлантический горбатый кит находятся под угрозой вымирания.

Мы присвоили уникальное значение атрибутам `id` тегов `title` и `desc` и присвоили эти `id` атрибуту `aria-labelledby` контейнера SVG. Экранные дикторы будут использовать их как узнаваемые, доступные имена.

Листинг 10.7 Обеспечение доступности контейнера SVG для людей с ограниченными возможностями (scatterplot.js)

Присвоить значение img атрибуту `role` контейнера SVG

```
const svg = d3.select("#scatterplot")
  .append("svg")
    .attr("viewBox", `0, 0, ${width}, ${height}`)
    .attr("role", "img")
    .attr("aria-labelledby", "scatterplotTitle
      scatterplotDescription");
svg
  .append("title")
    .attr("id", "scatterplotTitle")
    .text("Visualization of a collection of
      cetacean species");
svg
  .append("desc")
    .attr("id", "scatterplotDescription")
    .text("...");
```

Присвоить атрибуту `aria-labelledby` контейнера SVG список с идентификаторами тегов title и desc

Добавить тег title как первый дочерний элемент в контейнер SVG и присвоить ему уникальный идентификатор. Добавить короткий заголовок, кратко описывающий назначение графика

Добавить тег desc в контейнер SVG и присвоить ему уникальный идентификатор. Включить в него текст с описанием диаграммы, основными выводами, которые можно сделать, и значимыми тенденциями

Аналогичным образом оформим легенду. В листинге 10.8 мы добавляем заголовок **Legend for the area of the circles** (Легенда площадей кругов) и описание **Three circles representing a weight of 180, 90 and 10 tons** (Три круга представляют вес 180, 90 и 10 тонн). Также присвоим атрибуту `role` контейнера SVG значение `"img"`, а атрибуту `aria-labelledby` – уникальные идентификаторы тегов `<title>` и `<desc>`.

Листинг 10.8 Создание доступной легенды (legend.js)

```
const legendArea = d3.select(".legend-weight")
  .append("svg")
    .attr("width", 180)
    .attr("height", 120)
    .attr("role", "img")
    .attr("aria-labelledby", "legendAreasTitle
      legendAreasDescription");
```

Задать атрибуты `role` и `aria-labelledby` контейнера SVG

Добавить тег title как первый дочерний элемент в контейнер SVG и присвоить ему уникальный идентификатор. Добавить короткий заголовок с описанием

Добавить тег desc в контейнер SVG и присвоить ему уникальный идентификатор. Включить в него текст с описанием

```
legendArea
  .append("title")
    .attr("id", "legendAreasTitle")
    .text("Legend for the area of the circles");
legendArea
  .append("desc")
    .attr("id", "legendAreasDescription")
    .text("Three circles representing a weight of
      180, 90 and 10 tons.");
const sizes = legendArea
  .append("g")
    .attr("transform", "translate(0, 10)");
```

Для цветовой легенды статусов описание не требуется, поэтому в листинге 10.9 мы добавляем только заголовки статусов.

Листинг 10.9 Обеспечение доступности цветовой легенды (legend.js)

Задать атрибуты `role` и `aria-labelledby` каждого контейнера SVG

Добавить тег title как первый дочерний элемент в контейнер SVG и присвоить ему уникальный идентификатор. Добавить короткий заголовок с описанием

```
const legendConservationStatuses = statuses
  .append("svg")
    .attr("width", 32)
    .attr("height", 32)
    .attr("role", "img")
    .attr("aria-labelledby", d => `legendCS-${d.id}`);
legendConservationStatuses
  .append("title")
    .attr("id", d => `legendCS-${d.id}`)
    .text(d => `Color legend for the conservation
      status ${d.label}`);
legendConservationStatuses
  .append("circle")
    .attr("cx", 16)
    .attr("cy", 16)
    .attr("r", 15)
    ...
```

По завершении попробуйте озвучить проект с помощью экранного диктора. Закройте глаза, чтобы понять, как воспримет диаграмму человек с очень слабым зрением.

Визуализированные данные должны быть доступны для чтения экранному диктору

Пользователи экранных дикторов теперь могут получить общее представление о нашей визуализации и какой главный вывод можно сделать, глядя на нее. Но что, если они также захотят интерпретировать данные, прикрепленные к каждому кругу на диаграмме рассеяния?

Есть два основных способа сделать визуализированные данные доступными экранному диктору. Первый способ: не добавлять атрибуты `role` и `aria-labelledby` в контейнер SVG, в таком случае экранный диктор сможет озвучить содержимое текстовых элементов внутри SVG. Поскольку чтение текстовых элементов выполняется в том порядке, в каком они добавляются в DOM, нужно выбрать такой порядок, чтобы озвучивание содержимого этих элементов имело смысл для пользователей экранных дикторов. Если вы решите добавить текстовые элементы, доступные для экранных дикторов, но невидимые на экране, то присвойте их атрибутам `x` и `y` такие значения, чтобы визуально эти элементы оказались за пределами контейнера SVG.

Второй способ: добавить атрибут `desc`, который будет динамически получать желаемые значения данных. В листинге 10.10 объявляются функция с именем `generateDescription()` и массив, который сохраняется в константе с именем `desc`. В первый элемент этого массива помещается описание, которое мы написали в предыдущем разделе, содержащее самую важную вводную информацию. Затем выполняется обход данных и для каждого вида китообразных добавляется строка, состоящая из названия вида китообразных, его численности, размера, веса и природоохранного статуса. Функция `generateDescription()` вызывается при записи текста в атрибут `desc` контейнера SVG.

Листинг 10.10 Подготовка базовых данных для экранного диктора (scatterplot.js)

```
const generateDescription = () => {
  const desc = [];
  desc.push("... Introductory text ...");
  desc.push("Cetacean species:");
  data.forEach(d => {
    const cetacean = `
      ${d.common_name},
      population estimate:
      ➥ ${d3.format(".2s")
          (d.global_population_estimate)},
      maximum size: ${d.max_size_m} meter,
      maximum weight: ${d.max_weight_t} ton,
      conservation status: ${conservationStatuses
        .find(status =>
        ➥ status.id === d.status).label}
    `;
    desc.push(cetacean);
  });
  return desc;
};
```

Объявить пустой массив с именем `desc` и поместить в него текст описания, написанный ранее

Выполнить обход данных и для каждого вида китообразных создать строку с названием вида, его численностью, размером, весом и природоохранным статусом

```
svg
  .append("desc")
    .attr("id", "scatterplotDescription")
    .text(generateDescription());
```

Вызвать функцию generateDescription(), чтобы задать текст в атрибуте desc контейнера SVG

Если теперь прослушать новое описание, то вы услышите длинный список видов китообразных и связанных с ними данных. Этот способ больше подходит для визуализаций, содержащих ограниченное количество данных, но он делает доступными основные данные для программ чтения с экрана. В следующем разделе мы обсудим более удачное решение для диаграмм, содержащих много точек данных.

В качестве альтернативы должна предоставляться таблица данных

Несмотря на все усилия, иногда все же бывает сложно интерпретировать визуализации данных с помощью экранных дикторов. В качестве альтернативы рекомендуется предоставить удобочитаемую и доступную таблицу с данными, используемыми для создания визуализации. Эта стратегия полезна не только для пользователей экранных дикторов, но и для всех, кто захочет извлечь данные из проекта.

Как упоминалось ранее, этот подход лучше всего подходит для визуализаций, содержащих большой объем данных. В небольших визуализациях может быть достаточно предоставить данные в описании диаграммы.

Чтобы сделать таблицу данных доступной для пользователя, можно добавить две кнопки над диаграммой: одну для отображения диаграммы, а другую – таблицы, чтобы дать пользователям возможность выбора. Для этого, как показано в листинге 10.11, добавим эти кнопки как первые дочерние элементы в тег `<main>`. Первая кнопка будет иметь надпись **Display Chart** (Отобразить диаграмму) и атрибут `aria-pressed` со значением `true`, чтобы программы чтения с экрана могли сообщить пользователю, что эта кнопка нажата по умолчанию. Вторая кнопка будет имеет надпись **Display Data Table** (Отобразить таблицу данных) и атрибут `aria-pressed` со значением `false`. Мы также заключим легенду, кнопки фильтров и контейнер SVG в `div` с классом `"chart-container"` и добавим еще один элемент `div` с классом `"data-table-container"`.

Листинг 10.11 Добавление кнопок (index.html)

```
<body>
  <div class="container">
    <header>
      <h1>Cetacean species</h1>
      <div class="intro">...</div>
```

```
    </header>
    <main>
      <div class="mode-selector">
        <button
          type="button"
          aria-pressed="true"
            ➥ class="toggle toggle-chart active">
          Display Chart
        </button>
        <button
          type="button"
          aria-pressed="false"
            ➥ class="toggle toggle-table">
          Display Data table
        </button>
      </div>
      <div class="chart-container">
        <section>
          <div class="legend-container">
            ...
          </div>
        </section>
        <section>
          <div class="filters-container">
            ...
          </div>
          <div class="responsive-svg-container">
            <div id="scatterplot"></div>
          </div>
        </section>
      </div>
      <section>
        <div class="data-table-container"></div>
      </section>
    </main>
    <footer>
      ...
    </footer>
  </div>

  ...
</body>
```

Добавить в разметку две кнопки: одну для отображения диаграммы и одну для отображения таблицы данных. Первая кнопка нажата по умолчанию

Заключить легенду, фильтры и контейнер SVG в элемент div с именем класса "chart-container"

Создать новый элемент div с именем класса "data-table-container"

Теперь создадим новый файл с кодом на JavaScript и дадим ему имя `data-table.js`. В этом файле мы объявим функцию `createDataTable()`, которая будет получать набор данных, и вызовем ее из `load-data.js`, как только данные станут доступны. Внутри `createDataTable()` добавим элемент `table` внутрь `div` с классом `"data-table-container"`. Как показано в листинге 10.12, мы также добавили тег `caption`, как первый дочерний элемент таблицы, с текстом «Се-

tacean species data» (Данные о видах китов) и ссылку, чтобы пользователи могли загрузить набор данных. Позволить пользователям видеть данные в формате таблицы – это хорошо, но позволить им загрузить набор данных – еще лучше!

Далее добавим элемент строки таблицы (`tr`), в который добавим заголовок (`th`) для каждого столбца: «Common name» (Название вида), Population estimate (Оценка численности популяции), Maximum size (meter) (Максимальный размер (метров)), Maximum weight (ton) (Максимальный вес (тонн)) и Conservation status (Природоохранный статус). В каждый заголовок добавим атрибут области действия `col`, чтобы подсказать экранному диктору, какие ячейки относятся к этому заголовку. Наконец, выполним обход данных и добавим в таблицу строки с данными о китообразных (`td`).

Листинг 10.12 Создание таблицы данных (data-table.js)

```
const createDataTable = (data) => {

  const table = d3.select(".data-table-container")
    .append("table");

  table
    .append("caption")
      .text("Cetacean species data")
    .append("a")
      .attr("href", "./data/data.csv")
      .attr("download", "cetaceans_data")
      .text("Download the data in CSV format");

  const tableHeader = table
    .append("tr");
  tableHeader
    .append("th")
      .attr("scope", "col")
      .text("Common name");
  tableHeader
    .append("th")
      .attr("scope", "col")
      .text("Population estimate");
  tableHeader
    .append("th")
      .attr("scope", "col")
      .text("Maximum size (meter)");
  tableHeader
    .append("th")
      .attr("scope", "col")
      .text("Maximum weight (ton)");
  tableHeader
    .append("th")
      .attr("scope", "col")
      .text("Conservation status");
```

Добавить элемент table в div с классом "data-table-container"

Добавить элемент caption в таблицу и ссылку для загрузки набора данных

Добавить элементы заголовков столбцов и определить в них атрибут области действия `col`

```
  data.forEach(d => {
    const tableRow = table
      .append("tr");
    tableRow
      .append("td")
        .text(d.common_name);
    tableRow
      .append("td")
        .text(d3.format(".2s")
          (d.global_population_estimate));
    tableRow
      .append("td")
        .text(d.max_size_m);
    tableRow
      .append("td")
        .text(d.max_weight_t);
     tableRow
      .append("td")
        .text(conservationStatuses.find(status =>
          status.id === d.status).label);
  });
};
```

Выполнить цикл по данным и добавить в таблицу строку для каждого китообразного

Теперь нам осталось лишь обработать события щелчков на кнопках для отображения диаграммы или таблицы. Для этого, как показано в листинге 10.13, мы выбираем `div` с классом `"data-table-container"` и присвоим его стилевому свойству `display` значение `"none"`.

Затем добавим обработчик событий `click` для кнопки с классом `"toggle-table"` (вторая кнопка, добавленная ранее). При появлении события щелчка атрибуту `aria-pressed` этой кнопки присваивается значение `true` и в кнопку добавляется класс `"active"`. Для кнопки с классом `"toggle-chart"` выполняются обратные действия. Далее свойству `display` контейнера таблицы присваивается значение `"block"`, а тому же свойству контейнера диаграммы – значение `"none"`. Кроме того, в другую кнопку добавляется обработчик событий щелчка, производящий обратные действия: делает эту кнопку активной, а другую неактивной, скрывает таблицу и показывает диаграмму.

Листинг 10.13 Обработка щелчков на кнопках (data-table.js)

```
const createDataTable = (data) => {

  ...

  d3.select(".data-table-container")
    .style("display", "none");
```

Присвоить стилевому свойству display значение "none", чтобы скрыть таблицу

```
  d3.select(".toggle-table")
    .on("click", () => {
      d3.select(".toggle-table")
        .attr("aria-pressed", true)
        .classed("active", true);
      d3.select(".toggle-chart")
        .attr("aria-pressed", false)
        .classed("active", false);
      d3.select(".data-table-container")
        .style("display", "block");
      d3.select(".chart-container")
        .style("display", "none");
  });
```

Добавить обработчик событий щелчка в кнопку, отвечающую за отображение таблицы. При появлении события щелчка сделать эту кнопку активной, а другую неактивной, отобразить таблицу с данными и скрыть диаграмму

```
  d3.select(".toggle-chart")
    .on("click", () => {
      d3.select(".toggle-table")
        .attr("aria-pressed", false)
        .classed("active", false);
      d3.select(".toggle-chart")
        .attr("aria-pressed", true)
        .classed("active", true);
      d3.select(".data-table-container")
        .style("display", "none");
      d3.select(".chart-container")
        .style("display", "block");
  });

};
```

Добавить обработчик событий щелчка в кнопку, отвечающую за отображение диаграммы. При появлении события щелчка сделать эту кнопку активной, а другую неактивной, отобразить диаграмму и скрыть таблицу

На рис. 10.16 показаны кнопки выбора между диаграммой и таблицей. В данном случае была нажата кнопка таблицы и ниже отображается таблица с данными. Обратите внимание, что мы определили свои стили для кнопок и таблицы. Вы найдете их в `table.css`.

Таблицы данных могут оказаться более понятными для пользователей экранных дикторов, а также полезными для всех пользователей, желающих получить доступ к данным и просмотреть их. Уделите немного времени изучению проекта с расширением экранного диктора ChromeVox и сравните впечатления от прослушивания содержимого диаграммы и таблицы.

Пользовательские настройки клавиатуры и сенсорного управления не должны совпадать с комбинациями управления экранным диктором

Большинство экранных дикторов используют комбинации клавиш **Shift**, **Ctrl**, **Option**, стрелок и **Tab**, и очень важно не переопределить эти клавиши своими событиями в JavaScript. Переопределение этих событий может создать проблемы пользователям экранных дикторов.

Cetacean species

This visualization shows a collection of cetacean species. Each circle represents a cetacean. The circles' position on the scatterplot reveals their size and estimated population. The area of the circles is proportional to the weight of each cetacean. Their color represents the conservation status of each species.

Display Chart | Display Data table

Cetacean species data

Common name	Population estimate	Maximum size (meter)	Maximum weight (ton)	Conservation status
Blue whale	10k	33	173	Endangered
Fin whale	100k	27	80	Vulnerable
north Atlantic right whale	350	18	80	Critically Endangered
north Pacific right whale	1.3k	18	80	Endangered
southern right whale	14k	18	80	Least Concern
Sperm whale	1.1M	18	50	Vulnerable
Bowhead whale	10k	16	70	Least Concern
Sei whale	80k	16	30	Endangered
Bryde's whale	95k	15.5	26	Least Concern
Humpback whale	84k	15	30	Least Concern
Gray whale	21k	14	35	Least Concern
Antarctic minke whale	520k	10.2	15	Near Threatened
Common minke whale	200k	10	15	Least Concern
northern bottlenose whale	10k	9	7.5	Near Threatened
southern bottlenose whale	500k	7.5	8	Least Concern
Cuvier's beaked whale	100k	7	3	Least Concern
Beluga	140k	5	1.5	Least Concern
Narwhal	120k	5	1.6	Least Concern
Perrin's beaked whale	830	4.5	2	Endangered
Common bottlenose dolphin	600k	3.9	0.65	Least Concern
northern right whale dolphin	400k	3	0.1	Least Concern
White-beaked dolphin	100k	2.8	0.275	Least Concern
Australian snubfin dolphin	9.5k	2.6	0.15	Vulnerable
Rough-toothed dolphin	150k	2.6	0.15	Least Concern
Atlantic humpback dolphin	1.5k	2.5	0.15	Critically Endangered
Atlantic white-sided dolphin	250k	2.5	0.2	Least Concern
Baiji	7.0	2.5	0.16	Critically Endangered
south Asian river dolphin	3.5k	2.5	0.09	Endangered
Striped dolphin	2.0M	2.5	0.15	Least Concern
Pacific white-sided dolphin	1.0M	2.4	0.15	Least Concern
Pantropical spotted dolphin	3.0M	2.4	0.115	Least Concern
Atlantic spotted dolphin	100k	2.3	0.14	Least Concern
Dall's porpoise	1.1M	2.2	0.22	Least Concern
Harbour porpoise	700k	1.9	0.065	Least Concern
Hourglass dolphin	140k	1.8	0.12	Least Concern
Commerson's dolphin	22k	1.7	0.06	Least Concern
Franciscana	4.3k	1.7	0.053	Vulnerable
Hector's dolphin	7.4k	1.5	0.06	Endangered
Vaquita	12	1.5	0.055	Critically Endangered

Рис. 10.16 При нажатии соответствующей кнопки вместо диаграммы отображается таблица с данными. В ее заголовке присутствует ссылка для загрузки набора данных

Мы завершили приведение нашей визуализации в соответствие с рекомендациями по доступности для программ чтения с экрана. Теперь:

- разметка структурирована семантически;
- контейнер SVG предоставляет дополнительный контекст;
- визуализированные данные доступны для чтения экранным дикторам;
- в качестве альтернативы предоставляется таблица с данными;
- пользовательские настройки клавиатуры и сенсорного управления не переопределяют настройки программ чтения с экрана.

10.2.4 Взаимодействия

Пользователям с ограничениями моторики из-за инвалидности или временного состояния может быть сложно использовать мышь для навигации или взаимодействия с нашим контентом. Такие пользователи будут стремиться использовать клавиши со стрелками и **Tab**. В этом разделе мы обсудим, как сделать взаимодействие с диаграммой доступным для таких пользователей.

Пользователи должны иметь возможность осуществлять взаимодействия с помощью клавиатуры

В нашем проекте, когда указатель мыши оказывается над одним из кругов на диаграмме, отображается всплывающая подсказка с соответствующим видом китообразных. Если вы попробуете перемещаться по проекту с помощью клавиши **Tab** на клавиатуре, вы сможете поочередно посетить все кнопки и ссылки, но не круги на диаграмме. Чтобы наша визуализация была более доступна для людей с ограниченными возможностями, мы должны обеспечить возможность взаимодействий не только с помощью мыши, но и с помощью клавиатуры.

Мы можем разрешить элементам SVG принимать фокус, установив атрибут `tabindex`. Значением этого атрибута может быть любое целое число, определяющее порядок, в каком фокус ввода будет передаваться между элементами на странице. В следующем фрагменте показан код из `scatterplot.js`, который использует шаблон привязки данных для добавления кругов в диаграмму рассеяния. Мы сортируем данные о китообразных в порядке возрастания численности популяции и добавляем к кругам атрибут `tabindex` со значением `0`. Сортируя данные, мы выбираем порядок, в каком фокус ввода будет передаваться между кругами в DOM. С помощью атрибута `tabindex` мы сообщаем браузеру, что кругам можно передавать фокус ввода с помощью клавиши **Tab**:

```
innerChart
  .selectAll(".cetacean")
  .data(data.sort((a, b) => a.global_population_estimate -
   ➥ b.global_population_estimate))
  .join("circle")
    .attr("class", d => `cetacean cetacean-${d.status}`)
    ...
    .attr("tabindex", 0);
```

Затем, чтобы показать подсказку, нам нужно определить обработчик события `focus`, который вызывается, когда элемент получает фокус ввода, и обработчик события `blur`, вызываемый, когда элемент теряет фокус. В следующем фрагменте кода мы добавляем обработчик события `focus`, который отображает подсказку, и обработчик события `blur`, который скрывает подсказку. При подключении одного обработчика к нескольким событиям с помощью метода `on()` нужно разделить имена событий пробелом:

```
innerChart
  .selectAll(".cetacean")
  .data(data.sort((a, b) => a.global_population_estimate -
   ➥ b.global_population_estimate))
  .join("circle")
    .attr("class", d => `cetacean cetacean-${d.status}`)
    ...
    .attr("tabindex", 0)
    .on("mouseenter focus", showTooltip)
    .on("mouseleave blur", hideTooltip);
```

Вот и все! Теперь мы сможем получать подсказки, перемещаясь по элементам с помощью клавиатуры. Как отмечалось в главе 7, когда мы обсуждали реализацию фильтров над диаграммой рассеяния, какие-то круги могут покидать визуализацию, а другие – появляться в ней. Поскольку появляющиеся круги – это совершенно новые элементы DOM, мы должны повторить процесс сортировки данных, добавления атрибута `tabindex` и обработки событий `focus` и `blur`. В следующем фрагменте показаны изменения в `interactions.js`:

```
const updatedData = datum.id === "all"
  ? data
  : data.filter(d => d.hemisphere === datum.id)
        .sort((a, b) => a.global_population_estimate -
         ➥ b.global_population_estimate);
innerChart
  .selectAll("circle")
  .data(updatedData, d => d.uid)
  .join(
    enter => enter
      .append("circle")
```

```
...
.attr("tabindex", 0)
.on("mouseenter focus", showTooltip)
.on("mouseleave blur", hideTooltip)
...
```

Навигация и взаимодействия должны быть простыми и не требовать больших трудозатрат

Действия для получения всплывающих подсказок в нашей диаграмме рассеяния просты и не требуют больших трудозатрат. Однако при работе с маленькими кружками, представляющими мелкие виды китообразных, могут возникать некоторые сложности, особенно у людей с ограниченной подвижностью кистей рук, но все же способных в некоторой степени использовать мышь. Чтобы облегчить получение подсказок к маленьким кружкам, мы расположим под ними более крупные, но невидимые кружки, которые также будут захватывать события мыши и отображать подсказку.

В листинге 10.14 мы добавляем новый набор кругов позади исходных. Атрибуты `cx` и `cy` новых кругов совпадают с исходными, но их радиус увеличен до 30 пикселей. Атрибуту `fill` этих кругов мы присваиваем значение `"transparent"`. Поскольку мы лишь упрощаем взаимодействие с помощью мыши, то подключаем обработчик только к событиям `mouseenter` и `mouseleave` и показываем или скрываем подсказку соответственно. Обратите внимание, что мы по-прежнему принимаем события мыши и управления фокусом ввода в исходных кругах.

Листинг 10.14 Добавление невидимых кругов большего размера (scatterplot.js)

```
innerChart
  .selectAll("interaction-circles")
  .data(data)
  .join("circle")
    .attr("class", "interaction-circles")
    .attr("cx", d => xScale(d.global_population_estimate))
    .attr("cy", d => yScale(d.max_size_m))
    .attr("r", 30)
    .attr("fill", "transparent")
    .on("mouseenter", showTooltip)
    .on("mouseleave", hideTooltip);

innerChart
  .selectAll(".cetacean")
  .data(data.sort((a, b) => a.global_population_estimate -
  ➥ b.global_population_estimate))
  .join("circle")
    .attr("class", d => `cetacean cetacean-${d.status}`)
```

Добавить набор новых кругов позади исходных

Установить их радиус равным 30 пикселям и присвоить атрибуту `fill` значение "transparent"

Подключить обработчики событий мыши, отображающие и скрывающие всплывающую подсказку

```
.attr("cx", d => xScale(d.global_population_estimate))
.attr("cy", d => yScale(d.max_size_m))
.attr("r", d => rScale(d.max_weight_t))
.attr('fill-opacity', 0.6)
.attr("stroke", d => colorScale(d.status))
.attr("stroke-width", 2)
.attr("tabindex", 0)
.on("mouseenter focus", showTooltip)
.on("mouseleave blur", hideTooltip);
```

Теперь получить подсказку с помощью мыши гораздо проще. Всякий раз, реализуя обработку событий от мыши, важно охватить достаточно большую область, чтобы упростить взаимодействия для людей, испытывающих трудности с точным управлением мышью.

Предусмотреть подсказки, помогающие осуществлять взаимодействия

Как разработчики мы много внимания уделяем нашим проектам, продумываем, как и какие взаимодействия инициировать, поэтому все это кажется нам очевидным. Но для пользователей, впервые открывающих наш проект, дополнительные подсказки помогут выяснить, как взаимодействовать с диаграммой. Для этого можно просто добавить инструкции в верхнюю часть диаграммы.

В следующем фрагменте мы добавляем раздел инструкций над диаграммой в файле `index.html`. Как обсуждалось в разделе 10.2.1, мы должны убедиться, что текст имеет уровень сложности не выше 9:

```
<div class="filters-container">
  <h2>Select a group of cetaceans</h2>
     <div id="filters"></div>
</div>
<div class="instructions">Pass your mouse over the circles or access them
  with the tab key to discover the cetacean they represent.</div>
<div class="responsive-svg-container">
  <div id="scatterplot"></div>
</div>
```

Инструкции можно видеть на рис. 10.17. Обратите внимание, что мы также добавили пользовательские стили в `base.css`. Добавление инструкций – простой шаг, который может значительно улучшить опыт для всех ваших пользователей!

Мы завершили приведение взаимодействий с нашей визуализацией в соответствие с рекомендациями по доступности для пользователей мыши и клавиатуры:

- пользователи могут управлять взаимодействиями с помощью клавиатуры;

- навигация и взаимодействие просты и не требуют больших трудозатрат;
- предусмотрены подсказки для взаимодействий.

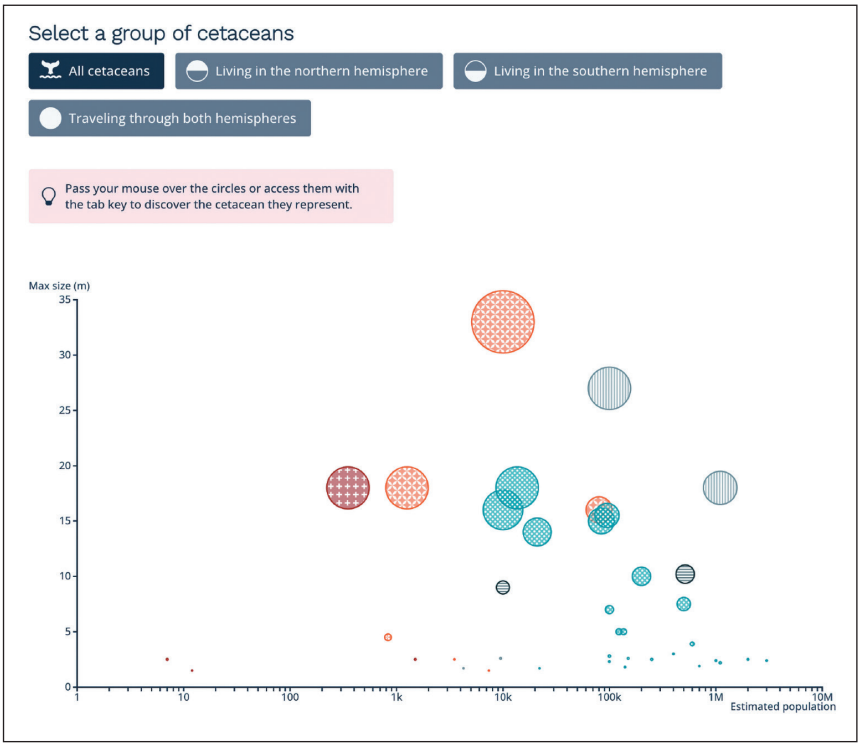


Рис. 10.17 Мы добавили инструкции над диаграммой, чтобы пользователи знали, как получить доступ к подсказкам

10.2.5 Прочее

В этом разделе мы обсудим еще три аспекта доступности, связанных с адаптивностью проекта, пользовательскими стилями и опасностями для лиц, страдающих эпилептическими припадками.

Проект должен обеспечивать эквивалентный опыт на экранах всех размеров

Обеспечение эквивалентного опыта на экранах всех размеров, также известное как адаптивный дизайн, имеет большое значение с точки зрения доступности. Мы должны гарантировать, что пользователи смогут взаимодействовать с нашим проектом, используя устройства, имеющиеся у них под рукой. В главе 9 мы подробно обсудили несколько стратегий адаптивной визуализации данных, таких как

использование макета сетки, изменение количества рисок на осях и ориентации диаграммы. Мы не будем обсуждать их здесь снова, а только отметим, что макет легенды этого проекта нужно адаптировать к небольшим экранам и обеспечить удобочитаемость диаграммы рассеяния.

Изменения стиля, внесенные пользователем, не должны отменяться

Некоторые пользователи изменяют стили для веб-страниц для удобства чтения. Они могут делать это с помощью расширений, таких как Stylebot, или импортировать свою таблицу стилей с помощью инструментов разработчика, имеющихся в браузере. Мы должны гарантировать, что наши стили не переопределяют стили, примененные пользователем. Это подразумевает проявление осторожности при работе со встроенными стилями, которые мы можем применить с помощью метода `style()` в D3. Встроенные стили отменяют большинство стилей CSS, определяемых в таблицах стилей. Использование их в элементах визуализации, для которых у пользователя, скорее всего, не будет своих предопределенных стилей, вполне допустимо. Но мы должны избегать определения стилей для общего макета страницы, таких как семейства, размеры и цвет шрифтов и т. д.

Визуальные эффекты не должны представлять опасности для лиц, страдающих эпилептическими припадками.

Некоторые цвета, такие как красный, и крупные узоры с параллельными полосами могут вызывать эпилептические припадки. Чтобы снизить риск припадков, избегайте ярких анимаций с красным цветом и не используйте красный цвет при отображении элементов, занимающих значительную часть экрана.

10.2.6 Дополнительные ресурсы

Вот некоторые ресурсы, посвященные вопросам доступности визуализаций данных:

- Dataviz Accessibility Resources: https://github.com/dataviza11y/resources;
- статья Сары Фосхайм (Sarah L. Fossheim) «Introduction to Accessible Data Visualizations with D3.js» (http://mng.bz/67Q5);
- статья Билла Шандера (Bill Shander) «Making Data Visualization Accessible» (http://mng.bz/oePj);
- MDN Web Docs, https://developer.mozilla.org/ru/docs/Learn/Accessibility.

Доступность визуализаций данных – обширная и важная тема, которая заслуживает отдельной книги. Мы обсудили некоторые из

наиболее важных аспектов доступности в этой главе, но вам еще многое предстоит узнать. После того как вы привыкнете автоматически применять приемы и рекомендации, описанные в этой главе, мы рекомендуем вам взять на вооружение все 57 рекомендаций по организации доступности, перечисленных в Chartability (https://chartability.fizz.studio/).

Как и в случае с адаптивными визуализациями, стратегии создания доступных визуализаций могут значительно различаться от проекта к проекту. Иногда даже будет иметь смысл добавить переключатель, который позволит пользователям выбирать между «обычным» и «доступным» режимами. Реализация этих стратегий потребует от нас приложения бóльших усилий, однако они могут иметь огромное значение для наших пользователей.

Интервью с Эстер Катона, доктором философии

Катона – научный сотрудник и доцент университета имени Лоранда Этвёша (Венгрия).

У вас богатый опыт в области информатики, статистики и социальных наук. Это нетрадиционное сочетание! Расскажите немного о своих интересах и о том, как они переплетаются.

Прежде всего спасибо, что выбрали нашу визуализацию для представления в книге и за возможность познакомить читателей со мной и нашей исследовательской группой.

По профессии я статистик, но мне нравится называть себя социальным информатиком. Я изучала социальные науки на бакалавриате BA в университете Лоранда Этвёша, и моими любимыми предметами в первом семестре были методология и социальная статистика. Я также изучала германистику, где впервые услышала о вычислительной лингвистике. Обработка естественного языка тогда еще не была широко распространена, но она сразу же заинтересовала меня. Затем я окончила магистратуру по статистике опросов и получила степень доктора философии по междисциплинарным социальным исследованиям. В круг моих исследовательских интересов входят: обработка естественного языка, методология исследований и визуализация данных.

Я с восхищением смотрю на ученых, которые тратят годы и десятилетия на изучение одной и той же области и по-настоящему глубоко вникают в нее, посвящая себя одной теме исследования. Я как методологический исследователь выбрала другой путь: я работаю с экспертами в предметной области, чтобы получить представление о широком круге областей. Я участвую в различных исследовательских проектах, таких как анализ форумов о депрессии, государственных закупок, речей политиков или свидетельств выживших в холокосте. В основном я занимаюсь кодированием, моделированием и визуализацией различных исследовательских проектов.

Я считаю, что будущее – за междисциплинарным сотрудничеством. Как социальный информатик я работаю над ответами на социальные вопросы, анализируя большие наборы данных и текстовые корпуса. Конечно, в поиске ответов, интерпретации и обобщении результатов визуализация играет ключевую роль.

Как вы пришли в сферу визуализации данных?

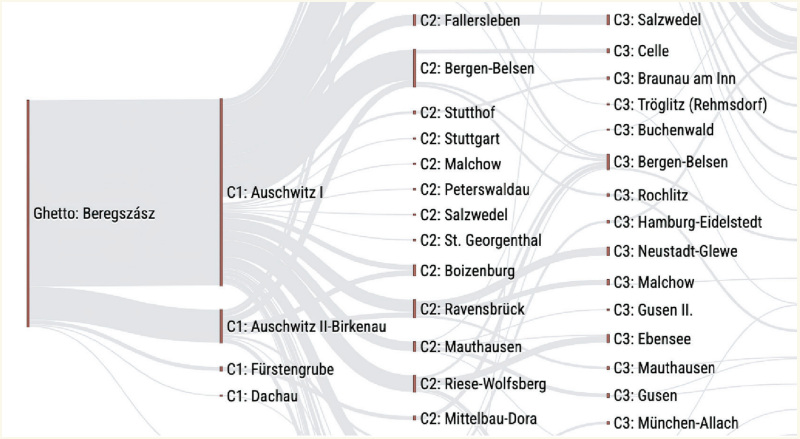
Первое мое знакомство с визуализацией данных состоялось во время стажировки в рекламном агентстве, где я работала младшим аналитиком данных. Мы использовали Power BI, но все, что я делала, было основано на интуиции. Я проработала в агентстве шесть месяцев и ушла оттуда, потому что не чувствовала, не видела в своей работе ничего захватывающего или творческого. После этого я нашла гораздо более захватывающую работу. Я стала частью отличной команды, где многому научилась: один из моих первых наставников в области науки о данных, Золтан Варжу (Zoltán Varjú), вовлекал меня в проекты, где я могла познакомиться с различными инструментами визуализации данных.

Я влюбилась в дизайн, цвета, типографику и графику. Если соединить все это с моей профессией, то получится моя главная страсть – визуализация данных. Почему-то долгое время визуализация данных была просто хобби, но в последние годы у меня стало появляться все больше возможностей применять на профессиональном уровне то, что я всегда любила делать.

Какие инструменты вы используете чаще всего для создания визуализаций данных? Какую роль в этом спектре инструментов вы отводите библиотеке D3?

В последнее время я чаще всего использую Tableau. Одна из мест моей работы – институт Government Transparency Institute (GTI), где я разрабатываю дашборды, отображающие данные о государственных закупках. Вот почему я часто обращаюсь к Tableau в своей академической работе. Это инструмент, который я преподаю на курсе бакалавриата по социологии, но для студентов магистратуры по статистике и анализу данных, где программирование является нормой, я преподаю пакет Altair для Python. Помимо Tableau и Altair, я также часто использую D3.js, потому что эта библиотека позволяет настраивать диаграммы без каких-либо ограничений. D3.js – отличный инструмент, который можно использовать для создания автономных визуализаций. Мне она нравится также тем, что позволяет легко экспортировать диаграммы и изменять их для публикации в статических изданиях, что очень здорово, поскольку большинство журналов публикуют статические диаграммы.

Представленный проект: ELTE Digital Lens (http://mng.bz/4JAg)



Крупный план диаграммы Сэнки, на которой изображены маршруты перемещения венгерских жертв холокоста

Ваш проект визуализирует маршруты перемещения евреев, депортированных из венгерского гетто Берегшас во время Второй мировой войны. Опишите этот проект и его контекст.

Наша исследовательская группа Digital Lens является частью исследовательского центра по социальной информатике в университете Лоранда Этвёша. Digital Lens – это междисциплинарная группа: мы используем вычислительные инструменты, обработку естественного языка, машинное обучение и визуализацию данных для анализа исторических источников. Мы исследуем письменные источники: ранние свидетельства евреев, выживших в холокосте, записанные Национальным комитетом по сопровождению депортированных (DEGOB – аббревиатура на Венгерском языке). Мы дополняем вычислительный анализ методами качественного анализа. Наши исследовательские вопросы затрагивают три аспекта: гендерный опыт, использование языка и географию. Мы также занимаемся всеобъемлющей темой этики в представлении холокоста. Визуализации, показанные в этом материале, фокусируются на географии.

Расскажите, как создавался этот проект.

Самой сложной частью проекта была предварительная обработка и стандартизация данных. Оригинальные протоколы были напечатаны на пишущих машинках и позднее оцифрованы (у нас есть отсканированное фото каждого протокола) и преобразованы в текстовый формат. Но процесс преобразования был выполнен неточно, требовались исправления.

Протоколы DEGOB состоят из двух частей: «жесткие» данные в заголовке и раздел интервью, описывающий опыт выживших. Первый содержит метаданные: демографические характеристики выживших, гетто, в котором они проживали, и лагеря, в которые их насильно вывезли. Раздел интервью описывает опыт выживших. Выбранные визуализации основаны на первой части протоколов, касающейся метаданных.

Прежде всего нам пришлось извлечь, проверить и исправить названия лагерей в протоколах, выбранных для визуализации. Это было необходимо в основном потому, что в разных протоколах использовались разные варианты написания, и в некоторых случаях названия лагерей, вероятно, были неправильно поняты и записаны с такими ошибками, что их невозможно было точно идентифицировать.

На этом этапе предварительной обработки данных, выполнявшемся вручную, нам помогало множество студентов.

Мне визуализация деревьев и диаграммы Сэнки кажутся особенно захватывающими, возможно потому, что история, которую они рассказывают, настолько очевидна. Что заставило вас выбрать эти диаграммы?

С самого начала мы знали, что будем использовать карты потоков, чтобы показать маршруты перемещения людей, переживших холокост, опираясь на стандартизированные и достоверные данные, извлеченные из протоколов.

Но я также хотела немного поэкспериментировать и создала диаграмму Сэнки, которая отображает количество выживших, перемещенных по тем же маршрутам. Узлы представляют лагеря, в которые они были депортированы.

Конечно, мы знаем, что в этом представлении отсутствует пространственность (у нас также есть карты, но они не были созданы в D3.js), но нашей целью было представить данные разными способами. Мы используем радиальное дерево, чтобы показать, насколько широко была распространена система лагерей. А с помощью диаграмм Сэнки показываем, что для многих выживших Освенцим не был конечным пунктом депортации, как это много раз предполагала историография. Более того, визуализация также показывает, что многие венгерские евреи были депортированы через другие лагеря. Диаграммы построены вокруг одной и той же темы: децентрализация Освенцима. Однако важно отметить, что здесь показаны только перемещения выживших; большинство венгерских евреев были убиты по прибытии в Освенцим.

Какими советами вы могли бы поделиться с желающими начать использовать D3 в академической среде?

Возможно, мой самый важный совет – осмельтесь сделать это! В моем сообществе Python и R – самые распространенные языки программирования; JS может показаться пугающим. Однако существует бесчисленное множество учебных пособий и руководств, вполне дружелюбных, которые помогут освоить этот язык. Кроме того, D3.js также считается хорошим выбором для ученых, поскольку большинство журналов предпочитают статические диаграммы. С этой точки зрения D3.js тоже является отличным инструментом, потому что позволяет экспортировать диаграммы в формат SVG, что даже дает возможность дополнительно отредактировать их в графических редакторах.

Итоги

- Мы должны стремиться предоставлять равноценный опыт пользователям с ограниченными возможностями.
- Чтобы сделать текстовый контент доступным, размер шрифта должен быть не менее 12 пикселей на экранах всех размеров.
- Необходимо объяснять назначение диаграммы и как ее читать.
- Сложность текста не должна превышать 9 баллов. Оценить сложность можно с помощью приложения Hemingway (https://hemingwayapp.com/).
- Все геометрические фигуры и крупный текст должны иметь контрастность по отношению к фону не менее 3:1, а обычный текст должен иметь контрастность не менее 4,5:1. Проверить контрастность можно с помощью WCAG Color contrast checker и WebAIM contrast checker (https://webaim.org/resources/contrastchecker/).
- Люди, страдающие цветовой слепотой, могут испытывать трудности с различением цветов в цветовой палитре. Поэтому рекомендуется использовать двойное кодирование (добавление узоров или использование различных форм в дополнение к цветам), чтобы информация передавалась не только с помощью цвета.
- Узоры SVG можно создавать с помощью элемента `<pattern>` внутри `<defs>`. Внутри `<pattern>` создается плитка, которая будет повторяться браузером по горизонтали и вертикали.
- Чтобы программы чтения с экрана (экранные дикторы) могли интерпретировать наш контент, мы всегда должны использовать семантические элементы HTML.
- Добавление элементов `<title>` и `<desc>` в контейнеры SVG, в дополнение к атрибуту `role` со значением `img`, является отличным способом помочь пользователям экранных дикторов получить больше информации о наших визуализациях. В качестве альтернативы следует также предоставить таблицу данных.
- Пользователи должны иметь возможность взаимодействовать с визуализацией с помощью клавиатуры. Фигуры SVG можно сделать доступными для навигации посредством клавиши **Tab**, настроив их атрибуты `tabindex`.
- Событие `focus` возникает, когда элемент получает фокус ввода, а событие `blur` – когда элемент теряет фокус ввода.
- Адаптивный дизайн важен с точки зрения доступности.
- Встроенные стили следует использовать с осторожностью, поскольку они могут переопределять таблицы стилей, загруженные пользователем.
- Мы должны осознавать риск возникновения эпилептических припадков из-за демонстрации красного цвета в визуализациях.

Часть III

Сложные визуализации данных

До сих пор в этой книге мы исследовали относительно простые типы визуализаций данных: столбчатые диаграммы, диаграммы рассеяния, кольцевые диаграммы и т. д. Но благодаря философии, лежащей в основе D3, с помощью этой библиотеки можно создавать гораздо более сложные визуализации. Самое замечательное, что при этом не нужно изучать новую библиотеку для сетей и еще одну для карт, потому что с тем же успехом можно использовать методы, с которыми мы познакомились к настоящему моменту.

В главе 11 мы возьмем иерархические данные о наиболее распространенных современных языках человеческого общения и визуализируем их с применением упакованной пузырьковой и древовидной диаграмм. В главе 12 мы рассмотрим динамику изменения взаимосвязей между персонажами пьесы «Ромео и Джульетта», используя сетевую визуализацию. Наконец, в главе 13 мы попробуем ответить на вопрос, насколько велик шанс родившегося в той или иной стране получить Нобелевскую премию, для чего воспользуемся картами.

11 Иерархические визуализации

В этой главе рассматриваются:

- конструирование корневого узла из иерархических данных;
- использование иерархических генераторов макетов;
- рисование упакованной пузырьковой диаграммы;
- рисование древовидной диаграммы.

В первых главах этой книги мы познакомились со способами визуализации числовых данных. Например, длина столбцов в столбчатой диаграмме в главе 3 представляла количество связанных ответов на опрос. Аналогично положение точек данных на линейной диаграмме в главе 4 отображало температуру. В этой главе мы займемся иерархическими визуализациями, которые кодируют отношения родитель–потомок и могут помочь выявить закономерности, которые остались бы незамеченными при использовании более простых визуализаций, с которыми мы работали до сих пор.

Иерархические визуализации передают отношения родитель–потомок через *вложение*, *связи* или *смежность*. На рис. 11.1 показаны два примера иерархических визуализаций, использующих вложение: упакованная пузырьковая диаграмма и древовидная карта. Как следует из названия, упакованная пузырьковая диаграмма изображает группу кругов (пузырьков). В ней имеется корневой элемент-родитель, самый внешний круг, и последующие уровни потомков,

называемые узлами. Все узлы-потомки «упакованы» в родительский узел, а размер кругов-родителей пропорционален количеству узлов-потомков, содержащихся в них. Размер листовых узлов – узлов-потомков на самом нижнем уровне может представлять произвольное свойство. Древовидная карта конструируется аналогично, но в ней вместо кругов используются вложенные прямоугольники. Древовидные карты более эффективно распределяют пространство, чем упакованные пузырьковые диаграммы, и их часто можно встретить в визуализациях, связанных с финансами.

Другой знакомый и интуитивно понятный способ визуализации отношений родитель–потомок – с использованием связей, как в древовидных диаграммах. Древовидные диаграммы могут быть линейными, как генеалогическое дерево, или радиальными, как показано на рис. 11.1. Линейные древовидные карты легче читать, но они могут занимать много места, тогда как радиальные деревья более компактны, но требуют больше усилий для расшифровки.

Наконец, иерархические закономерности можно визуализировать через смежность с помощью сосульчатых диаграмм, также называемых диаграммами слоев разделов. Такие диаграммы часто встречаются в информационных технологиях.

Диаграммы, представленные на рис. 11.1, выглядят по-разному, но строятся они с помощью D3 примерно одинаково – с применением функции генератора макета. В главе 5 мы узнали, что функции генераторов макетов в D3 добавляют информацию в существующий набор данных и эту информацию можно использовать для добавления желаемых фигур в контейнер SVG. Создание иерархических визуализаций осуществляется аналогично.

В этой главе мы построим две иерархические визуализации: упакованную пузырьковую диаграмму и линейную древовидную диаграмму. Наши визуализации будут основаны на наборе данных о 100 самых распространенных языках в мире. Готовые диаграммы вы можете увидеть по адресу http://mng.bz/QZye.

Каждый язык в нашем наборе данных принадлежит к группе (семейству) родственных языков, произошедших от общих предков. Семейства можно подразделить на более мелкие группы, называемые ветвями. Возьмем для примера пять самых распространенных языков. В табл. 11.1 видно, как можно хранить информацию о каждом языке в электронной таблице. Левый столбец содержит языки: английский, мандаринский китайский, хинди, испанский и французский. Следующие столбцы включают родственные семейства языков: индоевропейское и сино-тибетское, а также языковые ветви, германскую, синитскую, индоарийскую и романскую. Мы заполняем таблицу общим числом людей, владеющих языком, и числом носителей этого языка.

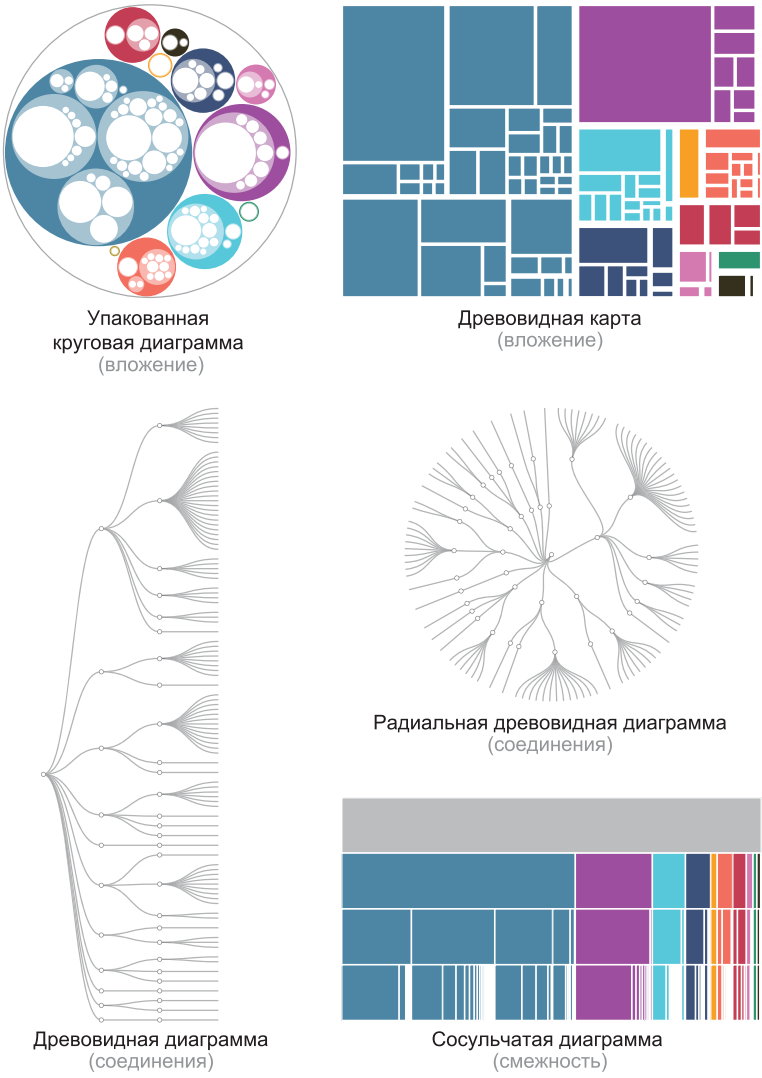


Рис. 11.1 В иерархических визуализациях отношения родитель–потомок могут передаваться посредством вложения (например, упакованная пузырьковая диаграмма или древовидная карта), связей (например, древовидная диаграмма) или смежности (например, сосульчатая диаграмма)

Таблица 11.1 Пять самых распространенных языков в мире

Язык	Семейство	Ветвь	Общее число владеющих языком	Число носителей
Английский	Индоевропейское	Германская	1132 млн	379 млн
Мандаринский китайский	Сино-тибетское	Синитская	1117 млн	918 млн
Хинди	Индоевропейское	Индоарийская	615 млн	341 млн
Испанский	Индоевропейское	Романская	534 млн	460 млн
Французский	Индоевропейское	Романская	280 млн	77 млн

Иерархические визуализации имеют один корневой узел, из которого исходят несколько ветвей, заканчивающихся *листьями*. В наборе данных, представленном в табл. 11.1, *корневой* узел можно назвать «Языки», как на рис. 11.2. Корень разделяется на два семейства языков – индоевропейское и сино-тибетское, – которые также разделяются на ветви: германскую, индоарийскую, романскую и синитскую. Наконец, справа располагаются листья: английский, хинди, испанский, французский и мандаринский китайский языки. Каждый язык, ветвь, семейство и корень называются *узлом*.

В первой половине этой книги мы в основном работали с проектами, имеющими традиционную структуру. Там главной целью было создать простую среду и сосредоточиться исключительно на D3. Но в проектах D3 обычно импортируются только отдельные модули JavaScript. Поэтому в этой главе мы модернизируем структуру нашего проекта и импортируем модули D3 по отдельности. Это сделает файлы проекта меньше, а значит, они будут загружаться быстрее, а кроме того, такой поход поможет нам увидеть, какие методы в каких модулях содержатся. Эти знания облегчат вам поиск в документации D3 в будущем.

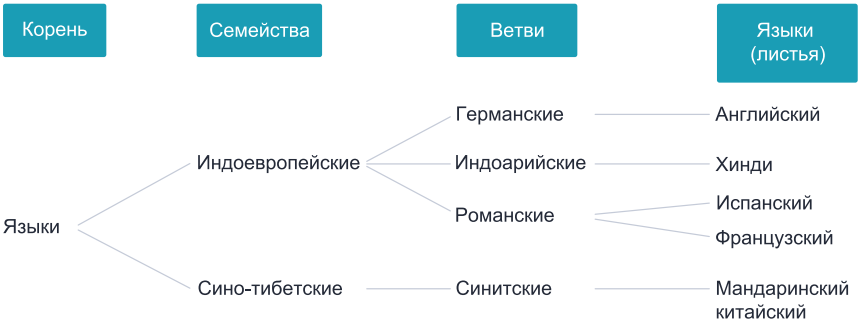


Рис. 11.2 Иерархическая визуализация имеет один корень, который разделяется на несколько ветвей, заканчивающихся листовыми узлами, которые в этом примере являются языками

Чтобы объединить наши файлы JavaScript и модули Node Package Manager (NPM) в один файл, который будет читать браузер, нам нужен упаковщик. Возможно, вы уже знакомы с Webpack или Rollup. Однако эти инструменты могут потребовать довольно много настроек, поэтому мы воспользуемся упаковщиком Parcel (https://parceljs.org/), очень простым в использовании и почти не требующим настройки.

Если Parcel еще не установлен на вашем компьютере, то вы можете установить его глобально с помощью следующей команды, где `-g` означает *global* – глобально. Запустите эту команду в окне терминала:

```
npm install -g parcel
```

Мы рекомендуем глобальную установку, чтобы сделать Parcel доступным для всех ваших проектов. Обратите внимание, что в зависимости от конфигурации компьютера в Mac и Linux может потребоваться добавить `sudo`, а в Windows – `runas` в начало команды.

После установки Parcel откройте папку `start` с файлами для этой главы в своем редакторе (http://mng.bz/X1PM). Если вы используете Visual Studio Code (VS Code), то откройте интегрированный терминал и выполните команду `npm install` для установки зависимостей проекта. На этом этапе единственные зависимости, которые у нас есть, позволят нам позже загрузить файлы данных CSV.

Чтобы запустить проект, выполните команду `parcel`, указав путь к корневому файлу:

```
parcel src/index.html
```

Введите адрес `http://localhost:1234/` в адресной строке вашего браузера, чтобы увидеть проект. Проект, отображаемый в браузере, будет автоматически обновляться при каждом сохранении файлов. После завершения рабочего сеанса вы можете остановить Parcel, нажав комбинацию **Ctrl**+**C** в окне терминала, где вы запускали проект.

В файле `index.html` мы уже загружаем файл `main.js` с тегом `script`. Поскольку мы будем работать с модулями, то присвоили атрибуту `type` в теге `script` значение `module`. Модули JavaScript имеют одну замечательную особенность – нам не нужно будет загружать дополнительные сценарии в `index.html`, все необходимое будет сделано в `main.js`. Обратите внимание, что нам также не понадобилось добавлять тег `script` для загрузки библиотеки D3. Мы установим и импортируем необходимые нам модули D3 в следующем разделе.

11.1 Форматирование иерархических данных

Для создания иерархических визуализаций D3 ожидает получить данные, отформатированные определенным образом. Нам доступны два основных варианта: файлы CSV или иерархические структуры JSON.

11.1.1 Файл CSV

В большинстве случаев данные представлены в табличной форме, часто в виде электронных таблиц. Такие файлы должны определять связи родитель–потомок с помощью столбцов. В табл. 11.2 мы реорганизовали наш набор данных с пятью самыми распространенными языками в столбцы «потомок» и «родитель». Мы будем использовать эти названия столбцов позже, чтобы сообщить D3, как устанавли-

вать отношения родитель–потомок. В первой строке, в столбце «потомок», определяется корневой узел «Языки». Поскольку это корневой узел, у него нет родителя. В следующих строках перечисляются прямые потомки корня: индоевропейское и сино-тибетское семейства. В обоих случаях в качестве родителя указан узел «Языки». Далее следуют языковые ветви (германская, синитская, индоарийская и романская) и объявляется, какое семейство языков является родительским. В конце таблицы находятся строки, в которых перечисляются сами языки (английский, мандаринский китайский, хинди, испанский и французский), и в качестве их родителей указывается языковая ветвь. Также в таблице присутствуют столбцы «Всего владеющих» и «Носителей», в которых для каждого языка указаны числа, которые мы будем использовать в наших визуализациях, но они не являются существенными для иерархических макетов.

Таблица 11.2 показывает структуру электронной таблицы перед построением иерархической визуализации с помощью D3. Мы экспортируем ее в виде CSV-файла и добавляем в наш проект. Обратите внимание, что вам не нужно создавать собственную электронную таблицу для выполнения упражнений в этой главе, готовый CSV-файл с именем `flat_data.csv`, содержащий информацию о 100 самых распространенных языках, вы найдете в папке `/data`.

Таблица 11.2 Отношения родитель–потомок, передаваемые через столбцы в CSV-файле

Потомок	Родитель	Всего владеющих	Носителей
Языки			
Индоевропейское	Языки		
Сино-тибетское	Языки		
Германская	Индоевропейское		
Синитская	Сино-тибетское		
Индоарийская	Индоевропейское		
Романская	Индоевропейское		
Английский	Германская	1132 млн	379 млн
Мандаринский китайский	Синитская	1117 млн	918 млн
Хинди	Индоарийская	615 млн	341 млн
Испанский	Романская	534 млн	460 млн
Французский	Романская	280 млн	77 млн

Давайте загрузим `flat_data.csv` в наш проект! Сначала создадим новый файл JavaScript в папке `/js`. Назовем его `load-data.js`, потому что именно здесь мы будем загружать набор данных. В листинге 11.1 мы создаем функцию с именем `loadCSVData()` и экспортируем ее, чтобы сделать доступной для других модулей JavaScript в нашем проекте.

Чтобы загрузить CSV-файл, нам нужно пойти другим путем, не так, как при использовании метода `d3.csv()`. Упаковщику Parcel

нужно передать правильный преобразователь для загрузки CSV-файла. Мы уже выполнили все необходимые настройки в проекте, установив модули, которые позволяют Parcel анализировать CSV-файлы (подробности вы найдете в файлах `.parcelrc` и `.parcel-transformer-csv.json`). Теперь нам осталось лишь загрузить CSV-файл вызовом функции `require()` и сохранить его в константе `csvData`. Если вывести константу `csvData` в консоль, то вы увидите, что она содержит массив объектов, каждый из которых соответствует строке в CSV-файле. Далее выполняется обход объектов в `csvData`, чтобы преобразовать в числа количества владеющих и носителей языка, и возвращается `csvData`.

Листинг 11.1 Загрузка CSV-файла в проект (load-data.js)

```
export const loadCSVData = () => {

  const csvData = require("../data/flat_data.csv");

  csvData.forEach(d => {
    d.total_speakers = +d.total_speakers;
    d.native_speakers = +d.native_speakers;
  });

  return csvData;

};
```

Импортировать flat_data.csv в проект. Вызовом функции require() мы сообщаем Parcel, что flat_data.csv – это внешний файл, который необходимо сохранить в папку dist

Преобразовать в числа количества владеющих и носителей языка

В `main.js` мы используем инструкцию `import`, чтобы получить доступ к функции `loadCSVData()`, как показано в листинге 11.2. Затем сохраняем массив, возвращаемый вызовом `loadCSVData()`, в константу с именем `flatData`.

Листинг 11.2 Импорт данных в main.js (main.js)

```
import { loadCSVData } from "./load-data.js";

const flatData = loadCSVData();
```

Импортировать функцию loadCSVData()

Сохранить массив, возвращаемый вызовом loadCSVData(), в константу с именем flatData

Наш следующий шаг – преобразовать CSV-данные в *иерархический* формат, т. е. в корневой узел с дочерними элементами. Модуль d3-hierarchy содержит метод `d3.stratify()`, который делает именно это. В нем также имеются другие методы, которые нам понадобятся для построения иерархических визуализаций.

Чтобы максимизировать производительность проекта, мы установим не всю библиотеку D3, а только необходимые модули. Начнем с `d3-hierarchy`. В VS Code откройте новое окно терминала и выполните следующую команду:

```
npm install d3-hierarchy
```

Затем создайте новый JavaScript-файл с именем `hierarchy.js`. В начале файла импортируйте метод `stratify()` из `d3-hierarchy`, как показано в листинге 11.3. Затем создайте функцию с именем `CSVToHierarchy()`, которая принимает CSV-данные. Обратите внимание, что мы делаем эту функцию общедоступной с помощью декларации экспорта.

Внутри `CSVToHierarchy()` мы объявляем генератор иерархии, вызывая метод `stratify()`. В традиционной организации проекта мы бы вызвали этот метод как `d3.stratify()`, но поскольку мы установили только модуль d3-hierarchy, для вызова функций больше не нужно использовать объект `d3` и мы можем считать `stratify()` обычной функцией.

Чтобы преобразовать CSV-данные в иерархическую структуру, функция `stratify()` должна знать, как определять отношение родитель–потомок. С помощью функции доступа `id()` мы указываем, по какому ключу можно найти дочерние элементы, в нашем случае `child` (дочерние элементы хранились в столбце `child` в исходном CSV-файле). С помощью функции доступа `parentId()` указываем, по какому ключу можно найти родительские элементы, в нашем случае `parent` (родительские элементы хранились в столбце `parent` в исходном CSV-файле).

Далее передаем данные в генератор иерархии и сохраняем получившуюся иерархическую структуру в константе с именем `root`. Эта вложенная структура данных имеет несколько методов, таких как `descendants()`, возвращающий массив всех узлов в дереве (`"Languages"`, `"Indo-European"`, `"Germanic"`, `"English"` и т. д.), и `leaves()`, возвращающий массив всех узлов, не имеющих потомков (`"English"`, `"Mandarin Chinese"`, `"Hindi"` и т. д.). Мы сохраняем узлы-потомки и листы в константы и возвращаем их с корневой структурой данных.

Листинг 11.3 Преобразование CSV-данных в иерархическую структуру (hierarchy.js)

```
import { stratify } from "d3-hierarchy";

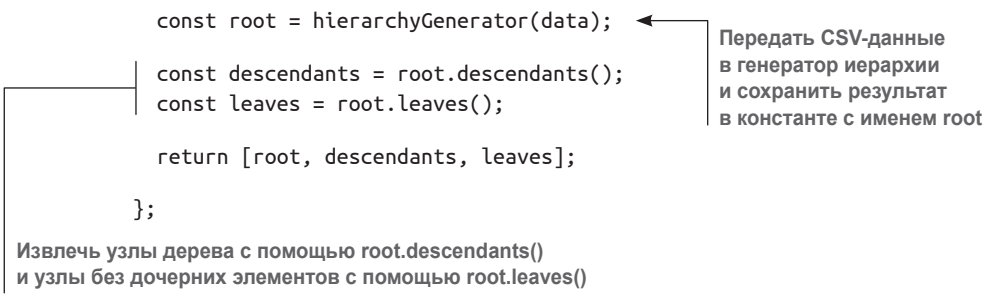
export const CSVToHierarchy = (data) => {

  const hierarchyGenerator = stratify()
    .id(d => d.child)
    .parentId(d => d.parent);
```

Импортировать функцию stratify() из модуля d3-hierarchy

Создать генератор иерархии с помощью функции stratify().

Использовать функции доступа id() и parentId(), чтобы указать, под какими ключами можно найти дочерние и родительские элементы в исходных CSV-данных



В `main.js` мы импортируем функцию `CSVToHierarchy()` и вызываем ее, чтобы получить корень, потомки и листья, как показано в листинге 11.4. Мы используем эту иерархическую структуру в следующих разделах для построения визуализаций.

Листинг 11.4 Импорт иерархических данных (main.js)

```
import { loadCSVData } from "./load-data.js";
import { CSVToHierarchy } from "./hierarchy.js";

const flatData = loadCSVData();
const [root, descendants, leaves] = CSVToHierarchy(flatData);
```

11.1.2 Файл JSON

Набор данных также может храниться в файле в иерархическом формате JSON, который изначально поддерживает иерархические структуры данных. Следующий объект JSON показывает пример структурирования данных. Корнем служит объект, заключенный в фигурные скобки ({}). Свойство `"name"` корня имеет значение `"Languages"`, а свойство `"children"` содержит массив объектов. Каждый прямой потомок корня – это семейство языков, которое содержит массив ветвей языков `"children"`, который, в свою очередь, включает массив `"children"` с листьями – названиями языков. Обратите внимание, что каждый потомок хранится внутри объекта родителя. В листовые объекты можно добавлять дополнительные данные, связанные с языками, такие как общее количество владеющих языком или носителей языка:

```
{
  "name": "Languages",
  "children": [
    {
      "name": "Indo-European",
      "children": [
        {
          "name": "Germanic",
          "children": [
```

```
            {
              "name": "English"
            }
          ]
        },
        {
          "name": "Indo-Aryan",
          "children": [
            {
              "name": "Hindi"
            }
          ]
        },
        {
          "name": "Romance",
          "children": [
            {
              "name": "Spanish"
            },
            {
              "name": "French"
            }
          ]
        },
      ]
    },
    {
      "name": "Sino-Tibetan",
      "children": [
        {
          "name": "Sinitic",
          "children": [
            {
              "name": "Mandarin Chinese"
            }
          ]
        }
      ]
    }
  ]
}
```

Иерархический файл JSON уже доступен в папке данных (`hierarchical-data.json`). Мы загрузим его в наш проект, действуя так же, как действовали, когда загружали файл CSV. Вернемся в файл `loaddata.js` и создадим функцию `loadJSONData()`, как показано в листинге 11.5. Она извлекает набор данных с помощью JavaScript-метода `require()`, сохраняет его в константе с именем `jsonData` и возвращает эту константу вызывающему коду.

Листинг 11.5 Загрузка данных JSON (load-data.js)

```
export const loadJSONData = () => {

  const jsonData = require("../data/hierarchical-data.json");
  return jsonData;

};
```

Вернемся в файл `main.js`, импортируем `loadJSONData()`, вызовем ее и сохраним возвращаемый ею объект в константе с именем `jsonData`, как показано в листинге 11.6.

Листинг 11.6 Импорт данных JSON в main.js (main.js)

```
import { loadCSVData, loadJSONData } from "./load-data.js";
import { CSVToHierarchy } from "./hierarchy.js";

const flatData = loadCSVData();
const [root, descendants, leaves] = CSVToHierarchy(flatData);

const jsonData = loadJSONData();
```

Чтобы сгенерировать иерархическую структуру данных из файла JSON, используем метод `d3.hierarchy()`. В листинге 11.7 мы импортируем функцию `hierarchy` из модуля d3-hierarchy, затем определяем функцию `JSONToHierarchy()`, которая принимает данные в формате JSON.

Внутри `JSONToHierarchy()` мы вызываем функцию `hierarchy()`, передаем ей исходные данные и сохраняем возвращаемую вложенную структуру данных в константе с именем `root`. Подобно структуре данных, полученной ранее с помощью функции `stratify()`, объект `root` имеет метод `descendants()`, возвращающий массив всех узлов дерева (`"Languages"`, `"Indo-European"`, `"Germanic"`, `"English"` и т. д.), и метод `leaves()`, возвращающий массив листьев, т. е. узлов, не имеющих потомков (`"English"`, `"Mandarin Chinese"`, `"Hindi"` и т. д.). Узлы и листья сохраняются в константы и возвращаются вместе с корневой структурой данных.

Листинг 11.7 Преобразование данных JSON в иерархию (hierarchy.js)

```
import { stratify, hierarchy } from "d3-hierarchy";

...

export const JSONToHierarchy = (data) => {

  const root = hierarchy(data);

  const descendants = root.descendants();
```

```
  const leaves = root.leaves();

  return [root, descendants, leaves];

};
```

Наконец, в `main.js` мы извлекаем структуры данных `root`, `descendants` и `leaves`, как показано в листинге 11.8. Чтобы не перепутать их со структурами, извлеченными из данных в формате CSV, мы добавляем в конец имен констант суффикс `_j`.

Листинг 11.8 Импорт иерархических данных в формате JSON (main.js)

```
import { loadCSVData, loadJSONData } from "./load-data.js";
import { CSVToHierarchy, JSONToHierarchy } from "./hierarchy.js";

const flatData = loadCSVData();
const [root, descendants, leaves] = CSVToHierarchy(flatData);

const jsonData = loadJSONData();
const [root_j, descendants_j, leaves_j] = JSONToHierarchy(jsonData);
```

> **ПРИМЕЧАНИЕ** В реальном проекте нет необходимости загружать данные сразу из двух форматов, CSV и JSON, обычно загрузка производится из какого-то одного формата. Здесь мы выполнили загрузку из двух форматов только в демонстрационных целях.

Загрузка иерархических данных в проекты D3 чаще всего осуществляется из файла CSV или иерархического файла JSON. Как показано на рис. 11.3, если данные загружаются из файла CSV, то они передаются в вызов `d3.stratify()`, генерирующий иерархическую структуру. Если данные загружаются из файла JSON, то они передаются в вызов `d3.hierarchy()`. Оба метода возвращают одну и ту же вложенную структуру данных, которой часто дают имя `root`. У этой структуры есть метод `descendants()`, возвращающий все узлы в иерархии, и метод `leaves()`, возвращающий узлы без потомков (листья).

Теперь, подготовив иерархические данные, можно переходить к самой интересной части – построению визуализаций! Именно этим мы и займемся в следующих разделах.

11.2 Построение упакованной пузырьковой диаграммы

В упакованной пузырьковой диаграмме каждый узел представлен кругом, а дочерние узлы вложены в родительский узел. Такая визуализация используется, когда нужно дать возможность одним взгля-

дом охватить общую иерархическую организацию. Она проста для понимания и имеет вполне привлекательный вид.

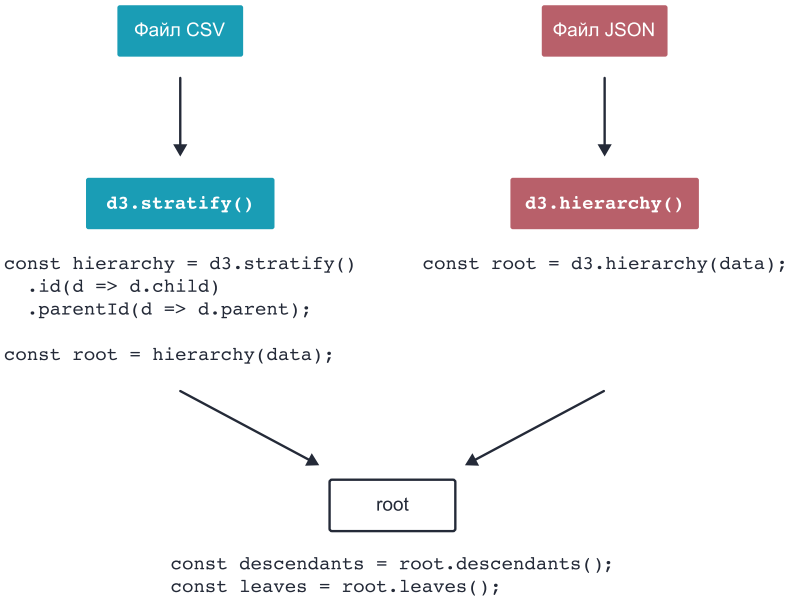


Рис. 11.3 Обычно исходные данные поступают к нам в виде файла CSV или JSON. Если данные загружаются из файла CSV, то они передаются в вызов `d3.stratify()`, генерирующий иерархическую структуру. Если данные загружаются из файла JSON, то они передаются в вызов `d3.hierarchy()`. Оба метода возвращают одну и ту же вложенную структуру данных, которую часто сохраняют в константе с именем root. У этой структуры есть метод `descendants()`, возвращающий все узлы в иерархии, и метод `leaves()`, возвращающий узлы без потомков (листья)

В этом разделе мы визуализируем наш набор данных со 100 наиболее распространенными языками в виде упакованной пузырьковой диаграммы, как показано на рис. 11.4. В данной визуализации внешний круг – это корневой узел, который мы назвали `"Languages"`. Круги более темного цвета – это семейства языков, а круги более бледного цвета – ветви языков. Белые круги обозначают языки, а их размеры представляют количество владеющих этим языком.

11.2.1 Создание макета упакованной пузырьковой диаграммы

Чтобы создать упакованную пузырьковую диаграмму с помощью D3, нужно использовать генератор макетов, подобный тем, что обсуждались в главе 5. Такие генераторы добавляют в существующий набор данных информацию, необходимую для построения диаграммы, например местоположение и радиус каждого круга и узла, как поясняется на рис. 11.5.

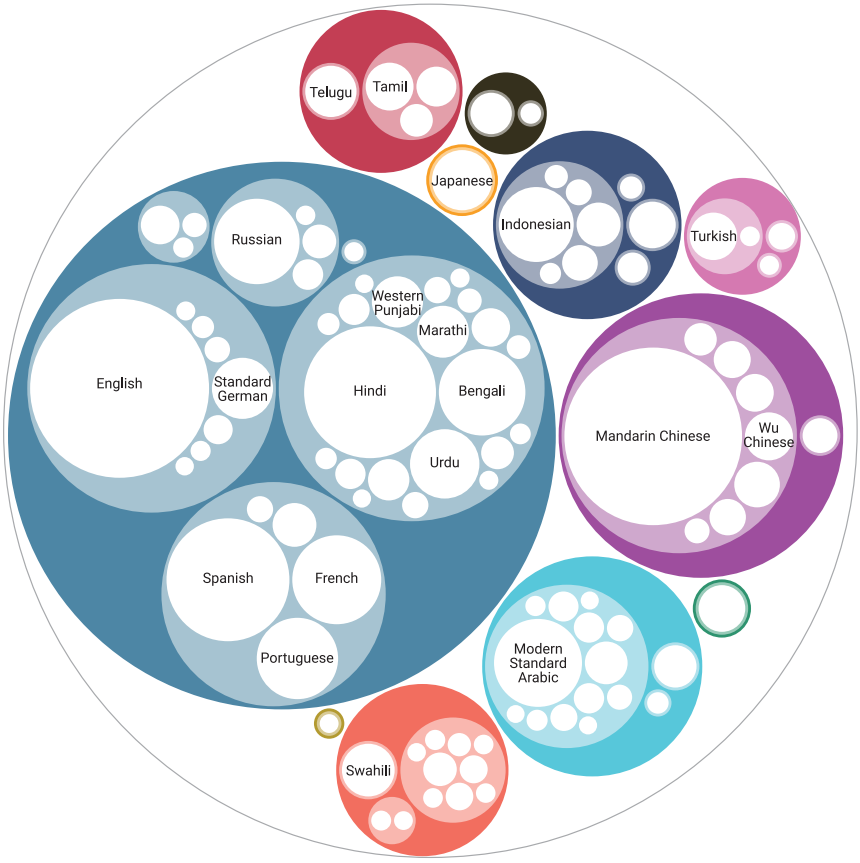


Рис. 11.4 Упакованная пузырьковая диаграмма, отображающая 100 самых распространенных языков

У нас уже есть иерархическая структура данных с именем `root`, и мы готовы перейти ко второму шагу в схеме на рис. 11.5. Сначала создадим в папке `js/` новый JavaScript-файл с именем `circle-pack.js` и объявим в нем функцию `drawCirclePack()`. Эта функция принимает структуру `root` и массивы `descendants` и `leaves`, созданные в предыдущем разделе:

```
export const drawCirclePack = (root, descendants, leaves) => {};
```

В `main.js` импортируем `drawCirclePack()` и вызовем ее, передав константы `root`, `descendants` и `leaves` в качестве аргументов:

```
import { drawCirclePack } from "./circle-pack.js";

drawCirclePack(root, descendants, leaves);
```

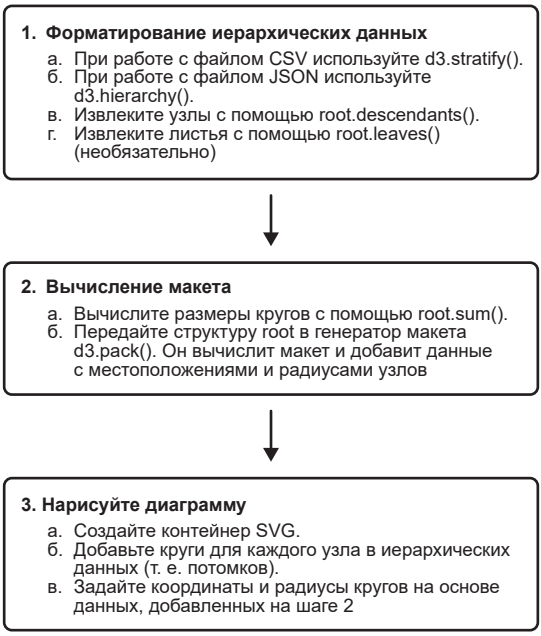


Рис. 11.5 Чтобы создать упакованную пузырьковую диаграмму с помощью D3, нужно прежде получить набор иерархических данных, которому часто дается имя `root`. Затем вызывается метод `sum()` структуры `root`, чтобы вычислить размеры кругов. Далее структура root передается в генератор макета `pack()`. Этот генератор добавляет координаты и радиус каждого круга в данные. Наконец, с использованием вновь сгенерированных данных круги добавляются в контейнер SVG

Вернемся в `circle-pack.js` и добавим внутрь функции `drawCircle-Pack()` код, вычисляющий макет. Как показано в листинге 11.9, сначала мы объявляем размеры диаграммы – 800 пикселей в ширину и в высоту. Затем объявляем объект `margin` с отступами, в котором задаем все четыре отступа (сверху, справа, снизу и слева) равными 1 пикселю. Нам понадобятся эти отступы позже, чтобы увидеть самый внешний круг. Наконец, вычисляем внутреннюю ширину и высоту диаграммы с помощью стратегии, описанной в главе 4.

Затем вызываем метод `sum()` объекта `root`. Этот метод отвечает за расчет агрегированного размера визуализации. Мы также указываем ключ, по которому должны вычисляться радиусы листовых узлов: `total_speakers`.

Для инициализации генератора макета пакета мы вызываем метод `pack()`, импортированный из модуля `d3-hierarchy` в начале файла. Мы используем его функцию доступа `size()`, чтобы задать общий размер макета, и функцию `padding()`, чтобы задать расстояние между кругами равным 3 пикселям.

Листинг 11.9 Вычисление макета упакованной пузырьковой диаграммы (circle-pack.js)

```
import { pack } from "d3-hierarchy";

export const drawCirclePack = (root, descendants, leaves) => {

  const width = 800;
  const height = 800;
  const margin = { top: 1, right: 1, bottom: 1,
    left: 1 };
  const innerWidth = width - margin.right - margin.left;
  const innerHeight = height - margin.top -
    margin.bottom;

  root.sum(d => d.total_speakers);

  const packLayoutGenerator = pack()
    .size([innerWidth, innerHeight])
    .padding(3);
  packLayoutGenerator(root);

};
```

Объявить константы с размерами диаграммы

Инициализировать генератор макета вызовом d3.pack(). Установить размеры с помощью констант innerWidth и innerHeight и задать отступ между кругами равным 3

Вызвать функцию генератора макета и передать структуру root в качестве аргумента

Вычислить агрегированную сумму всех узлов и сообщить d3, что размер конечных узлов должен быть пропорционален общему числу владеющих языком

Если вывести массив `descendants` в консоль, то можно увидеть, что генератор макета добавил в каждый узел следующую информацию:

- `id` – метка узла;
- `depth` – глубина, `0` для корневого узла, `1` для семейств языков, `2` для ветвей языков и `3` для языков;
- `r` – радиус круга узла;
- `x` – горизонтальная координата круга узла;
- `y` – вертикальная координата круга узла;
- `data` – объект, содержащий имя родительского узла, общее количество владеющих языком и количество носителей языка.

Эту информацию мы используем в следующем разделе для рисования диаграммы.

11.2.2 Рисование упакованной пузырьковой диаграммы

Теперь можно приступать к рисованию упакованной пузырьковой диаграммы! Чтобы выбрать и добавить элементы в DOM, нужно установить модуль d3-selection командой `npm install d3-selection`. Этот модуль содержит методы, отвечающие за управление деревом DOM, применение шаблона привязки данных и подключение обработчиков событий. В начале `circle-pack.js` импортируем функцию `select()` из `d3-selection`.

Внутри `drawCirclePack()` добавим контейнер SVG в `div` с идентификатором `"circle-pack"`, который уже существует в `index.html`. Настроим его атрибут `viewBox` и добавим группу для размещения внутренней диаграммы, следуя стратегии, описанной в главе 4.

Затем добавим круг для каждого узла в массиве `descendants`. Присвоим их атрибутам `cx` и `cy` значения `x` и `y`, добавленные в данные генератором макета пакета, как показано в листинге 11.10. Аналогично настроим атрибут радиуса `r`. Пока присвоим атрибуту `fill` кругов значение `"none"`, а атрибуту цвета обводки `stroke` – значение `"black"`. Чуть позже мы заменим эти значения.

Листинг 11.10 Рисование упакованной пузырьковой диаграммы (circle-pack.js)

```
import { pack } from "d3-hierarchy";
import { select } from "d3-selection";

export const drawCirclePack = (root, descendants, leaves) => {

  ...

  const svg = select("#circle-pack")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`)
      .append("g")
      .attr("transform", `translate(${margin.left},
        ${margin.top})`);

   svg
    .selectAll(".pack-circle")
    .data(descendants)
    .join("circle")
      .attr("class", "pack-circle")
      .attr("cx", d => d.x)
      .attr("cy", d => d.y)
      .attr("r", d => d.r)
      .attr("fill", "none")
      .attr("stroke", "black");

};
```

Добавить элемент SVG в div с идентификатором "circle-pack", который уже существует в index.html. Настроить его атрибут viewBox и добавить группу, которая будет содержать внутреннюю диаграмму

Используя шаблон привязки данных, добавить круг для каждого узла в массиве descendants

Настроить атрибуты координат центра и радиуса кругов, используя значения, добавленные генератором макета пакета

После этого шага упакованная пузырьковая диаграмма должна выглядеть, как показано на рис. 11.6. Наша визуализация обретает форму!

Нам нужно, чтобы каждое семейство языков имело свой цвет на диаграмме. Открыв файл `helper.js`, вы увидите массив с именем `languageFamilies` (см. фрагмент ниже). Он содержит список семейств языков и связанных с ними цветов. Этот массив можно использовать для создания цветовой шкалы, с помощью которой настраивать атрибут `fill` каждого круга:

```
export const languageFamilies = [
  { label: "Indo-European", color: "#4E86A5" },
  { label: "Sino-Tibetan", color: "#9E4E9E" },
  { label: "Afro-Asiatic", color: "#59C8DC" },
  { label: "Austronesian", color: "#3E527B" },
  { label: "Japanic", color: "#F99E23" },
  { label: "Niger-Congo", color: "#F36F5E" },
  { label: "Dravidian", color: "#C33D54" },
  { label: "Turkic", color: "#D57AB1" },
  { label: "Koreanic", color: "#33936F" },
  { label: "Kra-Dai", color: "#36311F" },
  { label: "Uralic", color: "#B59930" },
];
```

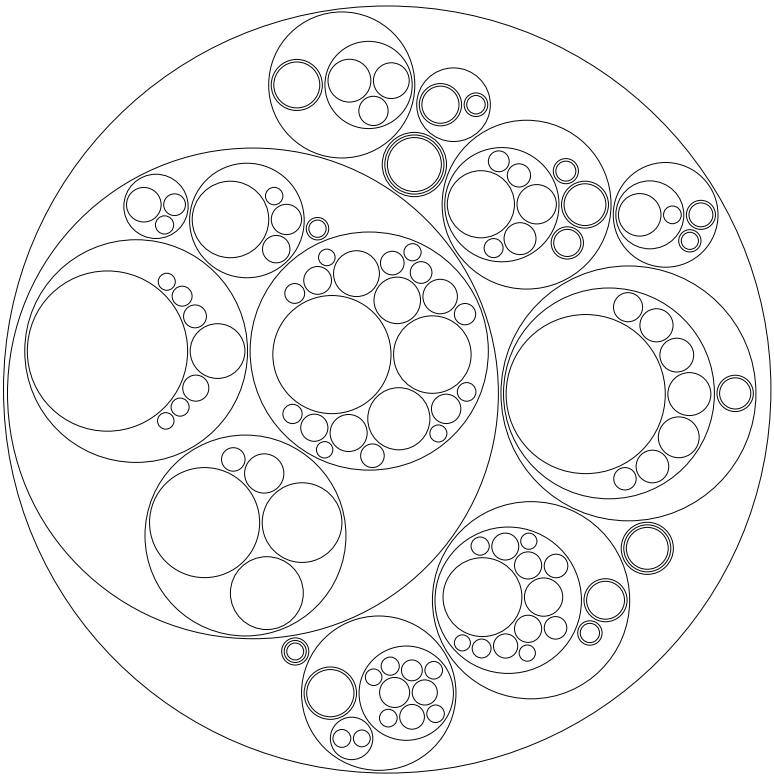


Рис. 11.6 Форма нашей упакованной пузырьковой диаграммы готова. Далее мы добавим цвет и надписи

Чтобы использовать шкалы D3, нужно установить модуль d3-scale командой `npm install d3-scale`. Для создания цветовой шкалы мы используем порядковую шкалу, которая принимает дискретный массив в качестве предметной области (т. е. семейства языков) и дискретный массив в качестве диапазона (т. е. цвета, соответствующие семействам). Создадим в папке `js/` новый файл с именем `scales.js`

и добавим в него код, показанный в листинге 11.11. В начале файла импортируем `scaleOrdinal` из `d3-scale` и массив `languageFamilies` из `helper.js`. Затем мы объявим порядковую шкалу `colorScale`, передав ей массив названий семейств языков в качестве предметной области и массив цветов в качестве диапазона. Эти массивы генерируются с помощью JavaScript-метода `map()`.

Листинг 11.11 Создание цветовой шкалы (scales.js)

```
import { scaleOrdinal } from "d3-scale";
import { languageFamilies } from "./helper";

export const colorScale = scaleOrdinal()
  .domain(languageFamilies.map(d => d.label))
  .range(languageFamilies.map(d => d.color));
```

В конце раздела 11.2.1 мы обсудили, как генератор макета D3 добавил дополнительную информацию в набор данных `descendants` (т. е. узлов), включая их глубину. На рис. 11.7 показано, что наша упакованная пузырьковая диаграмма имеет глубину от 0 до 3. Самый внешний круг, представляющий корневой узел `"Languages"`, имеет глубину 0, серую обводку и прозрачную заливку. Следующие круги представляют семейства языков с глубиной 1. Их атрибуту `fill` присваивается цвет, возвращаемый только что объявленной цветовой шкалой. Далее следуют ветви языков, имеющие глубину 2. Они наследуют более светлую версию цвета родителя. Наконец, листовые узлы, или языки, имеют глубину 3 и белый цвет. Эта цветовая градация не следует какому-то конкретному правилу и направлена на то, чтобы сделать отношения родитель–потомки как можно более явными.

Вернемся в файл `circle-pack.js` и присвоим атрибутам `fill` кругов цвет, получаемый вызовом цветовой шкалы. В начало файла импортируем цветовую шкалу, созданную ранее в `scales.js`. Для получения более светлых цветов для ветвей языков (кругов с глубиной 2) используем метод `interpolate`, доступный в модуле d3-interpolate. Установите этот модуль командой `npm install d3-interpolate` и импортируйте метод в начале `circle-pack.js`.

В листинге 11.12 мы возвращаемся к коду, настраивающему атрибут `fill` кругов. Для оценки глубины каждого узла используется оператор `switch()`. Если глубина равна 1, то узел является семейством языков. Мы передаем идентификатор `id` шкале и получаем соответствующий цвет. Для ветвей языков мы тоже вызываем цветовую шкалу, но передаем значение `id` родительского узла (`d.parent.id`). Затем передаем цвет, возвращаемый шкалой, функции `interpolate()`. В первом аргументе мы передаем полученный цвет, а во втором – белый цвет `"white"`, т. е. цвет на другом конце интерполиру-

емого диапазона. Мы также передаем значение `0.5`, чтобы указать, что нам нужно значение, находящееся посередине между исходным и белым цветом. Наконец, мы возвращаем белый цвет `"white"` для всех оставшихся узлов.

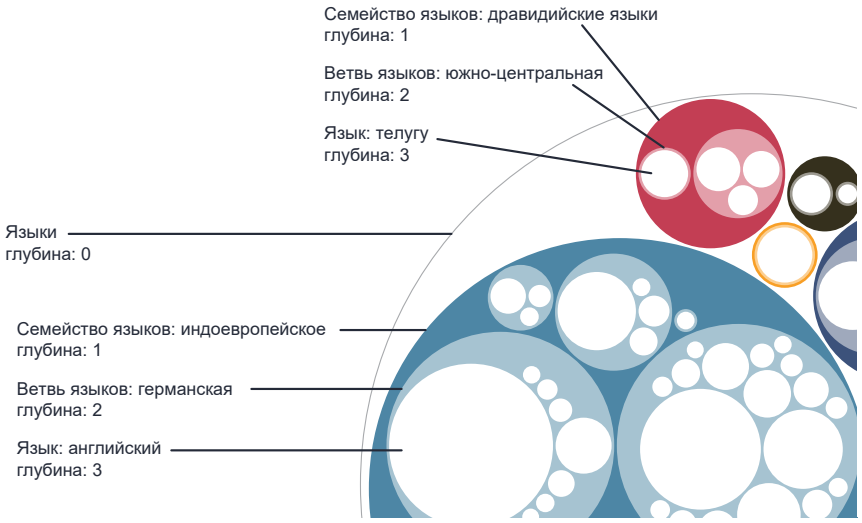


Рис. 11.7 Самый внешний круг на нашей диаграмме – это корневой узел «Languages» с глубиной 0. Далее следуют семейства языков с глубиной 1, ветви языков с глубиной 2 и собственно языки с глубиной 3

Мы также меняем значение атрибута `stroke` кругов. Если глубина равна 0, т. е. узел является корневым, то возвращается серый цвет. В противном случае для этого атрибута возвращается значение `"none"`.

Листинг 11.12 Вычисление цвета кругов (circle-pack.js)

```
...
import { colorScale } from "./scales";
import { interpolate } from "d3-interpolate";

export const drawCirclePack = (root, descendants, leaves) => {

  ...
  svg
    .selectAll(".pack-circle")
    .data(descendants)
    .join("circle")
      .attr("class", "pack-circle")
      ...
      .attr("fill", d => {
        switch (d.depth) {
```

Проверить глубину каждого узла с помощью оператора switch()

```
      case 1:
        return colorScale(d.id);
      case 2:
        return interpolate(colorScale(
          d.parent.id), "white")(0.5);
      default:
        return "white";
    };
  })
  .attr("stroke", d => d.depth === 0 ? "gray" : "none");

};
```

Если глубина равна 1, то это узел семейства языков. Цвет заливки определяется вызовом цветовой шкалы

Если глубина равна 2, то это узел ветви языка. Цвет заливки вычисляется путем интерполяции между цветом родителя (семейства языков) и белым цветом

Все остальные узлы получают белую заливку

Если узел имеет глубину 0, то для соответствующего круга задается серая обводка. В противном случае обводка не отображается

По завершении диаграмма должна выглядеть так, как показано на рис. 11.8.

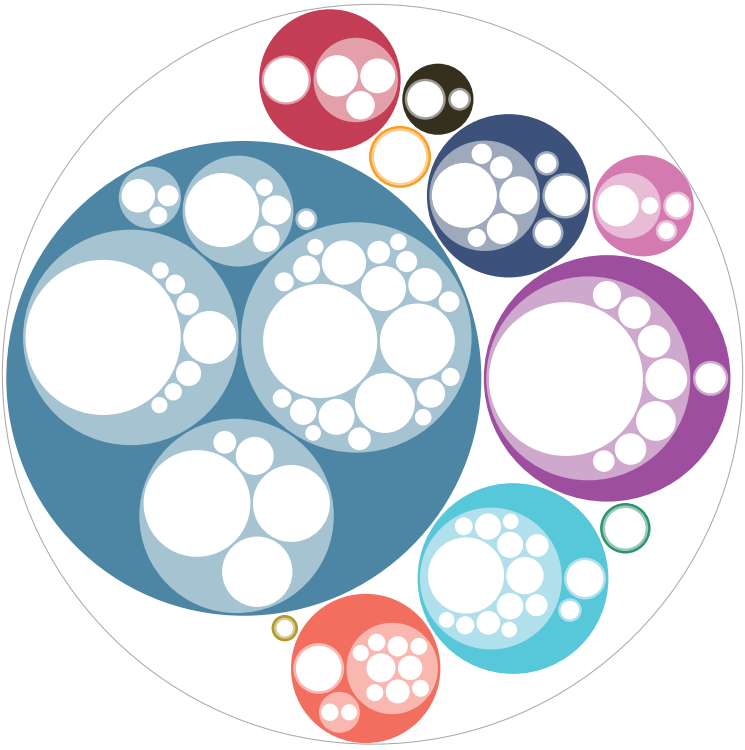


Рис. 11.8 Упакованная пузырьковая диаграмма после добавления цветов

11.2.3 Добавление меток

Наша упакованная пузырьковая диаграмма выглядит достаточно привлекательно, но в ней отсутствуют какие-либо подсказки о том, какие языки, ветви и семейства представляют те или иные круги.

Один из главных недостатков упакованной пузырьковой диаграммы – сложность размещения читаемых меток. Но так как наш проект цифровой, мы можем предоставить читателю дополнительную информацию через взаимодействия с мышью.

В этом разделе сначала добавим метки к большим кругам. Затем создадим интерактивный инструмент, предоставляющий дополнительную информацию при наведении указателя мыши на тот или иной листовой узел.

Добавление меток с использованием HTML-элементов div

В данной визуализации мы имеем дело и с относительно короткими названиями языков, такими как «французский» или «немецкий», и с довольно длинными, такими как «современный стандартный арабский» или «западный пенджаби». Чтобы отобразить эти длинные названия в пределах соответствующих им кругов, нужно разбить их на несколько строк. Но, как вы наверняка помните из предыдущих обсуждений, реализовать это с применением текстовых элементов SVG непросто и требует много работы. Намного проще работать с обычным текстом HTML, который автоматически разбивается на строки при необходимости! А теперь угадайте, как мы поступим? Верно! Мы будем использовать самые обычные элементы HTML внутри элемента SVG!

SVG-элемент `foreignObject` позволяет включать обычные HTML-элементы в контейнер SVG, например `div`. Этот `div` можно стилизовать как любой другой `div`, и его текст автоматически будет разбиваться на строки по мере необходимости.

Элементы `foreignObject` добавляются так же, как любые другие. Внутрь этих элементов можно добавить нужные нам `div`. Проще говоря, элемент `foreignObject` можно рассматривать как шлюз между мирами SVG и HTML.

Чтобы не перегружать диаграмму информацией, мы добавим метки не во все, а только в самые крупные круги. В листинге 11.13 мы сначала определяем минимальный радиус кругов, в которые добавим метки (здесь 22 пикселя). Затем используем шаблон привязки данных, чтобы добавить элемент `foreignObject` в каждый листовой узел, радиус которого больше выбранного минимального. Элементы `foreignObject` имеют четыре обязательных атрибута:

- `width` – ширина элемента, соответствующая диаметру круга (`2 * d.r`);
- `height` – высота элемента, здесь высота устанавливается равной 40 пикселям, чего достаточно для отображения трех строк текста;

- `x` – горизонтальная координата верхнего левого угла, мы выбираем ее так, чтобы этот угол соответствовал левой стороне круга (`d.x - d.r`);
- `y` – вертикальная координата верхнего левого угла, мы выбираем ее так, чтобы этот угол находился на 20 пикселей выше центра круга, поскольку общая высота элемента `foreignObject` равна 40 пикселям (`d.y - 20`).

Затем нужно указать имя и пространство имен XML элемента, который будет добавлен в `foreignObject`. Вот почему мы добавляем `xhtml:div`, а не просто `div`. Мы добавляем в этот элемент `div` класс `"leaf-label"` и в качестве текстового содержимого назначаем идентификатор узла. Файл `visualization.css` уже содержит стили, необходимые для центрирования метки по горизонтали и вертикали внутри элемента `foreignObject`.

Листинг 11.13 Добавление меток (circle-pack.js)

```
export const drawCirclePack = (root, descendants, leaves) => {

  ...

  const minRadius = 22;
  svg
    .selectAll(".leaf-label-container")
    .data(leaves.filter(leave => leave.r ===
      minRadius))
    .join("foreignObject")
      .attr("class", "leaf-label-container")
      .attr("width", d => 2 * d.r)
      .attr("height", 40)
      .attr("x", d => d.x - d.r)
      .attr("y", d => d.y - 20)
    .append("xhtml:div")
      .attr("class", "leaf-label")
      .text(d => d.id);
};
```

Использовать шаблон привязки данных, чтобы добавить элемент foreignObject в каждый круг с радиусом больше минимального

Установить атрибуты `width`, `height`, `x` и `y` элемента foreignObject. Обратите внимание, что элементы с шириной и высотой больше, чем у foreignObject, не будут видны

Добавить элемент div внутрь foreignObject, указав пространство имен ("xhtml:div"). Добавить в него класс с именем "leaf-label", а в качестве текстового содержимого назначить идентификатор узла

После добавления меток диаграмма должна выглядеть, как показано на рис. 11.4 или в онлайн-версии проекта (http://mng.bz/QZye). Теперь можно найти основные языки в визуализации.

Предоставление дополнительной информации с помощью всплывающих подсказок

В главе 7 мы обсуждали, как обрабатывать события мыши с помощью D3, например для отображения всплывающих подсказок. В этом разделе мы реализуем нечто похожее, но на этот раз подсказки будут

отображаться не поверх визуализации, а сбоку. Это отличный вариант, когда нужно показать больше, чем несколько строк информации. Поскольку такая подсказка создается с помощью элементов HTML, ее также легко будет стилизовать.

В файле `index.html` раскомментируйте `div` с идентификатором `"info-container"`. Этот `div` содержит два основных элемента:

- элемент `div` с идентификатором `"instructions"`, который отображается по умолчанию и сообщает пользователям, что они могут навести указатель мыши на круги, чтобы получить дополнительную информацию;
- `div` с идентификатором `"info"`, который при наведении указателя мыши на конкретный круг будет отображать информацию о языке: название, ветвь, семейство и количество владеющих им. Этот `div` также имеет класс `"hidden"`, который присваивает значение `0` свойствам `max-height` и `opacity`, делая элемент невидимым. Соответствующие стили вы найдете в `visualization.css`.

Снова вернемся к `circle-pack.js`. В начале файла, как показано в листинге 11.14, мы импортируем функцию `selectAll()` из `d3-selection`. Также нужно установить модуль d3-format и импортировать его функцию `format()`.

Чтобы различать уровни узлов, в листинге 11.14 значения их глубины добавляются к именам классов. Затем с помощью функции `selectAll()` выбираются все круги с классом `"pack-circle-depth-3"` и все элементы `foreignObject`. К листовым узлам и их меткам мы подключаем обработчик события `mouseenter` вызовом метода `on()`. В функции обратного вызова этого обработчика используем данные, прикрепленные к элементу, и заполняем подсказку о соответствующем языке, ветви, семействе и количестве говорящих. Обратите внимание, что для отображения количества говорящих с тремя значащими цифрами и суффиксом (`".3s"`), например `"M"` для обозначения «миллионов», мы используем функцию `format()`.

Затем мы скрываем `div` с идентификатором `"instructions"` и показываем подсказку, добавляя и удаляя имя класса `"hidden"`. Мы также добавляем обработчик события `"mouseleave"`, в котором скрываем подсказку и показываем инструкции.

Листинг 11.14 Обработка событий мыши (circle-pack.js)

```
import { select, selectAll } from "d3-selection";
import { format } from "d3-format";

export const drawCirclePack = (root, descendants, leaves) => {
```

```
  ...

  svg
    .selectAll(".pack-circle")
    .data(descendants)
    .join("circle")
      .attr("class", d => `pack-circle
        pack-circle-depth-${d.depth}`)
      ...

  selectAll(".pack-circle-depth-3, foreignObject")
   .on("mouseenter", (e, d) => {

    select("#info .info-language").text(d.id);
    select("#info .info-branch .information")
    ➥   .text(d.parent.id);
    select("#info .info-family .information")
    ➥   .text(d.parent.data.parent);
    select("#info .info-total-speakers .information")
    ➥      .text(format(".3s")(d.data.total_speakers));
    select("#info .info-native-speakers .information")
    ➥      .text(format(".3s")(d.data.native_speakers));

    select("#instructions").classed("hidden", true);
    select("#info").classed("hidden", false);

   })
   .on("mouseleave", () => {
    select("#instructions").classed("hidden", false);
    select("#info").classed("hidden", true);

   });

};
```

Выбрать все элементы с классом pack-circle-depth-3 и элементы foreignObject. Добавить обработчик события помещения указателя мыши над кругом или одним из foreignObject с меткой

Добавить в каждый круг имя класса, содержащее глубину круга

Добавить обработчик события помещения указателя мыши над кругом или меткой

Заполнить элемент div информацией о языке, ветви, семействе и количестве говорящих

Скрыть инструкции и показать подсказку

Добавить обработчик события, когда указатель мыши покидает круг или метку

Показать инструкции и скрыть подсказку

Теперь при наведении указателя мыши на узел языка справа от диаграммы должна появиться дополнительная информация о языке, ветви, семействе и количестве носителей языка, как показано на рис. 11.9.

Недостаток упакованных пузырьковых диаграмм – они плохо подходят для отображения информации на экранах мобильных устройств. Да, круги по-прежнему позволяют четко видеть отношения родитель–потомок, но сделать метки хорошо читаемыми намного сложнее. Кроме того, круги, соответствующие языкам, могут стать слишком маленькими, поэтому использование события касания для отображения информации может вызывать сложности. Чтобы обойти эти недостатки, для мобильных устройств можно выбрать стековую диаграмму или другой тип визуализации.

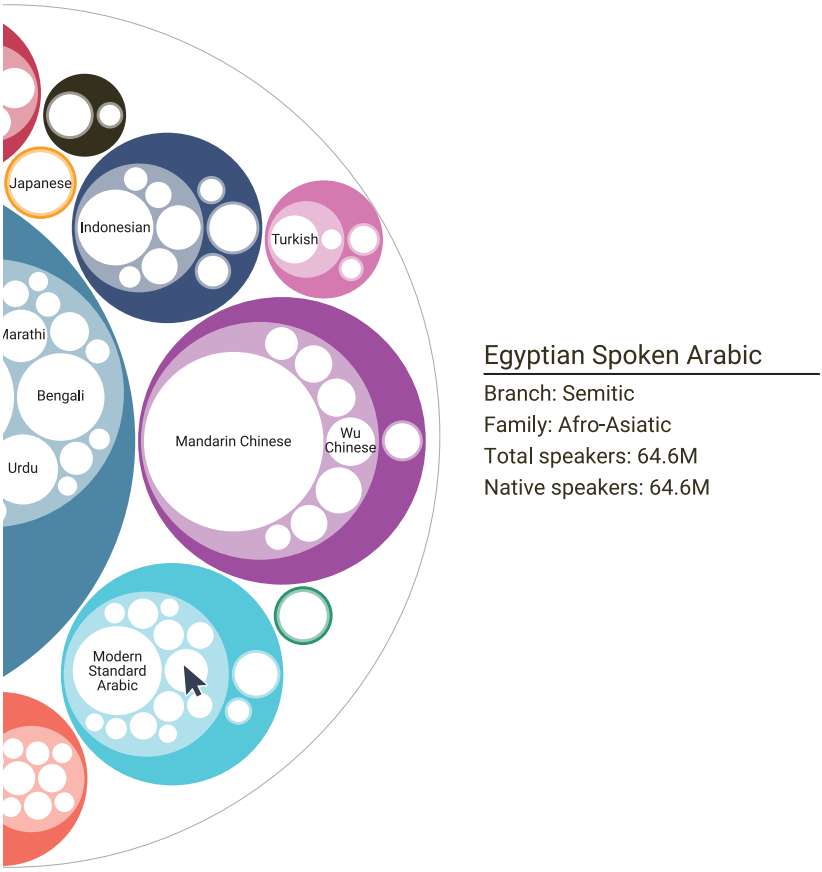


Рис. 11.9 При наведении указателя мыши на узел языка справа от диаграммы отображается информация о названии языка, ветви, семействе и количестве носителей

11.3 Построение древовидной диаграммы

Часто для визуализации отношений родитель–потомок используют древовидные диаграммы. Они похожи на генеалогические деревья. Как и упакованные пузырьковые диаграммы, они состоят из узлов, но также явно показывают связи между ними. В этом разделе мы построим древовидную диаграмму для 100 самых распространенных языков (см. рис. 11.10). Слева – корневой узел `"Languages"`. Он делится на семейства языков, которые, в свою очередь, подразделяются на ветви и, наконец, языки. Общее количество говорящих на каждом языке мы отобразим размером круга, а цветом обозначим принадлежность языка к тому или иному семейству.

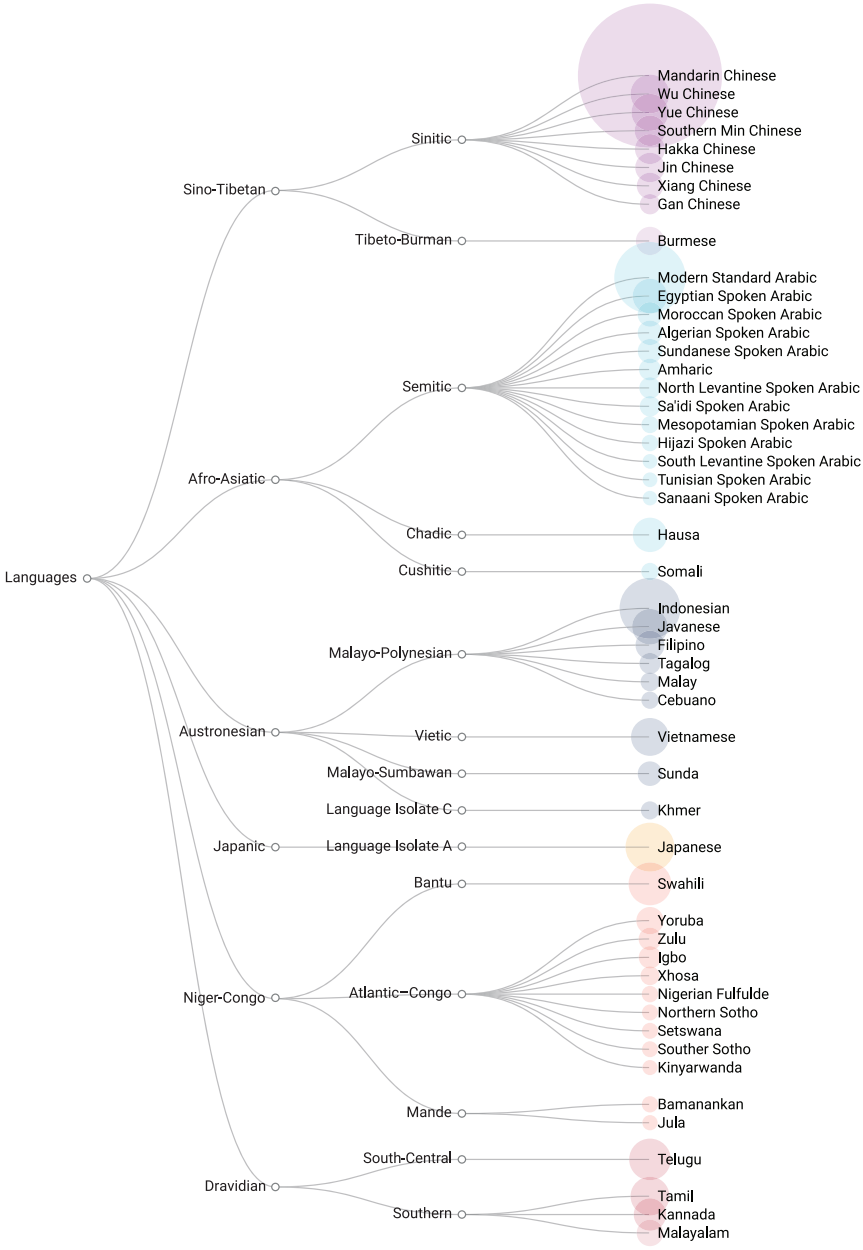


Рис. 11.10 Часть древовидной диаграммы, которую мы построим в этом разделе

11.3.1 Создание древовидного макета

Подобно упакованной пузырьковой диаграмме, созданной в предыдущем разделе, древовидные диаграммы тоже создаются с по-

мощью генератора макета, как описано на рис. 11.11. Генератор макета `d3.tree()` является частью модуля d3-hierarchy и генерирует информацию, которую можно использовать для рисования связей и узлов.

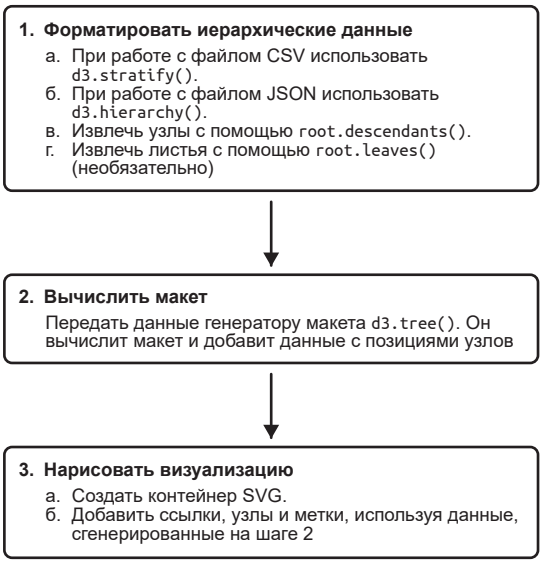


Рис. 11.11 Чтобы создать древовидную диаграмму с помощью D3, нужно прежде всего получить набор иерархических данных, которому часто дается имя `root`. Далее структура `root` передается в генератор макета `tree()`. Этот генератор добавляет координаты каждого узла в данные. Наконец, с использованием вновь сгенерированных данных в контейнер SVG добавляются ссылки, узлы и метки и задаются их позиции и размеры

Чтобы сгенерировать древовидный макет, начнем с создания нового файла и назовем его `tree.js`. Внутри этого файла создадим функцию `drawTree()`, которая принимает иерархические данные (`root`, `descendants` и `leaves`) в качестве параметров. В листинге 11.15 мы объявляем диаграмму, задаем ее ширину 1200 пикселей и высоту 3000 пикселей. Обратите внимание, что высота пропорциональна количеству конечных узлов в диаграмме и была подобрана методом проб и ошибок. Начиная работу с древовидной визуализацией, начните с приблизительного значения и последовательно корректируйте его, наблюдая, как визуализация выглядит на экране.

Чтобы получить генератор макета, вызовем функцию `tree()`, импортированную из d3-hierarchy в начале файла, и добавим функцию доступа `size()`, которая принимает массив с шириной и высотой диаграммы. Нам нужно, чтобы дерево развертывалось слева направо, поэтому в первом аргументе мы передаем `innerHeight`, а во втором `innerWidth`. Мы передали бы эти аргументы в другом порядке,

если бы нам нужно было, чтобы дерево развертывалось сверху вниз. Наконец, мы вызываем полученный генератор макета и передаем ему иерархические данные (`root`).

Листинг 11.15 Генерирование древовидного макета (tree.js)

```
import { tree } from "d3-hierarchy";

export const drawTree = (root, descendants, leaves) => {

  const width = 1200;
  const height = 3000;
  const margin = {top:60, right: 200, bottom: 0,
    left: 100};
  const innerWidth = width - margin.left -
    margin.right;
  const innerHeight = height - margin.top -
    margin.bottom;

  const treeLayoutGenerator = tree()
    .size([innerHeight, innerWidth]);
  treeLayoutGenerator(root);

};
```

Установить размеры диаграммы

Объявить генератор древовидного макета с помощью функции tree() и передать размеры диаграммы в функцию доступа size(). В первом аргументе мы передаем innerHeight, а во втором innerWidth, чтобы дерево развертывалось слева направо

Передать иерархические данные (root) в генератор

В `main.js` нам также необходимо импортировать функцию `drawTree()` и передать ей в аргументах `root`, `descendants` и `leaves`:

```
import { drawTree } from "./tree.js";

drawTree(root, descendants, leaves);
```

11.3.2 Рисование древовидной диаграммы

После того как макет будет сгенерирован, останется лишь нарисовать древовидную диаграмму, что делается довольно просто. Как обычно, сначала нужно добавить контейнер SVG и задать его атрибут `viewBox`. В листинге 11.16 мы добавляем этот контейнер в `div` с идентификатором `"tree"`, который уже существует в `index.html`. Обратите внимание, что для этого мы импортировали функцию `select()` из модуля `d3-selection` в начале файла. Мы также добавляем группу SVG в контейнер и смещаем ее в соответствии с заданными отступами, следуя стратегии, описанной в главе 4.

Для создания связей нам понадобится функция генератора связей `d3.link()`. Она работает точно так же, как генератор линий, представленный в главе 3, и является частью модуля d3-shape, который нужно установить командой `npm install d3-shape`. В начале

файла мы импортируем функцию `link()` из `d3-shape`, а также функцию `curveBumpX()`, которую будем использовать для определения формы связей.

Затем мы объявляем генератор связей с именем `linkGenerator`, передавая функцию `curveBumpX` в вызов `link()`. Дополнительно устанавливаем его функции доступа `x()` и `y()` для использования значений, хранящихся в ключах `y` и `x`. Так же, как при подготовке генератора макета дерева, значения `x` и `y` передаются в противоположном порядке, чтобы дерево рисовалось слева направо, а не сверху вниз.

Для рисования связей используем шаблон привязки данных, чтобы добавить элементы `path` на основе данных из `root.links()`. Этот метод возвращает массив связей в дереве с их исходными и конечными точками. Затем вызывается генератор связей для вычисления атрибута `d`. В завершение созданные связи стилизуются и устанавливается степень их непрозрачности равной 60 %.

Листинг 11.16 Рисование связей (tree.js)

```
...
import { select } from "d3-selection";
import { link, curveBumpX } from "d3-shape";

export const drawTree = (root, descendants) => {

  ...

  const svg = select("#tree")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`)
    .append("g")
      .attr("transform", `translate(${margin.left},
        ${margin.top})`);

  const linkGenerator = link(curveBumpX)
    .x(d => d.y)
    .y(d => d.x);
  svg
    .selectAll(".tree-link")
    .data(root.links())
    .join("path")
      .attr("class", "tree-link")
      .attr("d", d => linkGenerator(d))
      .attr("fill", "none")
      .attr("stroke", "gray")
      .attr("stroke-opacity", 0.6);

};
```

Добавить контейнер SVG в div с идентификатором "tree". Установить его атрибут viewBox и добавить элемент группы, который будет действовать как внутренняя диаграмма

Объявить генератор связей вызовом функции d3.link() и передать ему в качестве функции кривой функцию curveBumpX. Настроить функции доступа x() и y() так, чтобы они извлекали значения ключей y и x, добавленных ранее генератором макета дерева. Обратите внимание, что x и y здесь меняются местами, чтобы получить дерево, расширяющееся слева направо, а не сверху вниз

Использовать шаблон привязки данных для добавления элемента path, описывающего каждую связь, возвращаемую методом root.link(). Вызвать генератор связей, чтобы установить атрибут d

Теперь связи будут выглядеть, как показано на рис. 11.12. Обратите внимание, что на этом рисунке показана только часть дерева, потому что оно очень высокое!

Чтобы выделить каждый узел, отобразим их в виде кругов. Маленькие круги с серой обводкой будут представлять корень, семейства, ветви языков, а круги, соответствующие самим языкам, будут иметь размеры, пропорциональные общему числу владеющих языком, и цвет, соответствующий семейству.

Для вычисления размера кругов, представляющих языки, нам понадобится шкала. В листинге 11.17 показан фрагмент из файла `scales.js`, в котором мы импортируем `scaleRadial()` из `d3-scale`. Предметная область шкалы непрерывна и простирается от 0 до максимального количества людей, говорящих на одном из языков. Диапазон тоже непрерывный и изменяется в пределах от 0 до 83 пикселей – радиус наибольшего круга из упакованной пузырьковой диаграммы, созданной в предыдущем разделе.

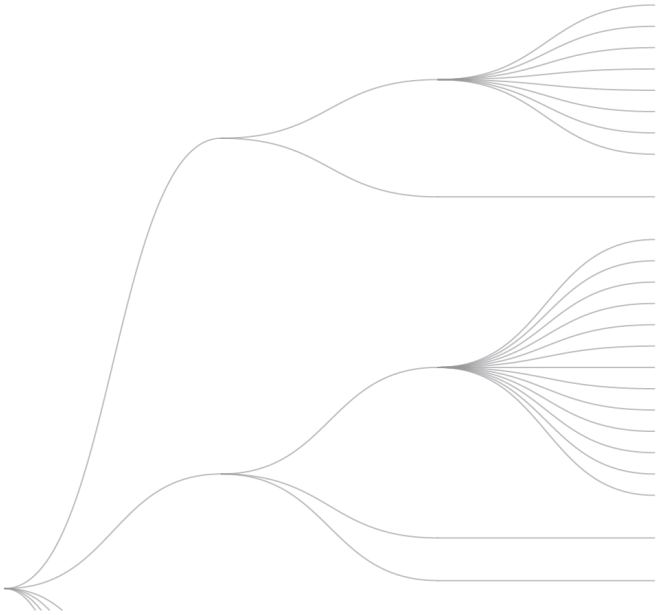


Рис. 11.12 Вид связей в древовидной диаграмме

Поскольку максимальное количество говорящих на языке доступно только после извлечения данных и создания иерархии (`root`), мы должны заключить радиальную шкалу в функцию с именем `getRadius()`. Когда понадобится вычислить радиус круга, мы просто передадим количество говорящих на данном языке, а также максимальное количество говорящих среди всех языков, и эта функция вернет радиус.

Листинг 11.17 Извлечение значений из радиальной шкалы (scales.js)

```
import { scaleOrdinal, scaleRadial } from "d3-scale";
...

export const getRadius = (maxSpeakers, speakers) => {
  const radialScale = scaleRadial()
    .domain([0, maxSpeakers])
    .range([0, 83]);

  return radialScale(speakers);
};
```

Вернемся в `tree.js` и вычислим максимальное количество владеющих некоторым языком с помощью метода `d3.max()`. Чтобы использовать этот метод, нужно установить модуль d3-array командой `npm install d3-array` и импортировать функцию `max()` в начале файла. Мы также импортируем функцию `getRadius()` и цветовую шкалу из `scales.js`.

Затем, как показано в листинге 11.18, используем шаблон привязки данных, чтобы для каждого дочернего узла добавить круг во внутреннюю диаграмму. Атрибутам `cx` и `cy` кругов присваиваются соответствующие значения с помощью ключей `x` и `y`, добавленных генератором макета. Если круг является конечным узлом, то его радиус задается пропорциональным количеству владеющих этим языком вызовом функции `getRadius()`, цвет задается с помощью цветовой шкалы, непрозрачность заливки `fill-opacity` устанавливается равной 30 %, а атрибуту `stroke` присваивается `"none"`. Для других кругов устанавливаются радиус 4 пикселя, белая заливка и серая обводка.

Листинг 11.18 Добавление узлов (tree.js)

```
...
import { max } from "d3-array";
import { getRadius, colorScale } from "./scales";

export const drawTree = (root, descendants) => {
  ...

  const maxSpeakers = max(leaves, d => d.data.total_speakers);
  svg
    .selectAll(".node-tree")
    .data(descendants)
    .join("circle")
      .attr("class", "node-tree")
      .attr("cx", d => d.y)
      .attr("cy", d => d.x)
```

Рассчитать максимальное количество говорящих на некотором языке с помощью метода d3.max()

Использовать шаблон привязки данных, чтобы добавить элемент круга для каждого узла. Установить координаты центра с помощью ключей `x` и `y`, добавленных генератором макета дерева

Если круг является листовым узлом, то установить радиус с помощью getRadius(), передав максимальное количество говорящих и количество говорящих на текущем языке. Иначе установить радиус равным 4 пикселям

```
    .attr("r", d => d.depth === 3
      ? getRadius(maxSpeakers, d.data.total_speakers)
      : 4
    )
```

Если круг является листовым узлом, установить его атрибут `fill` с помощью цветовой шкалы. В противном случае установить белый цвет

```
    .attr("fill", d => d.depth === 3
      ? colorScale(d.parent.data.parent)
      : "white"
    )
```

Если круг является листовым узлом, установить его атрибут `fill-opacity` равным 30 %, а атрибут `stroke` равным "none". В противном случае установить `fill-opacity` равным 100 %, а атрибут `stroke` – равным "gray"

```
    .attr("fill-opacity", d => d.depth === 3
      ? 0.3
      : 1
    )
    .attr("stroke", d => d.depth === 3
      ? "none"
      : "gray"
    );
};
```

Чтобы завершить древовидную диаграмму, добавим метку для каждого узла. В листинге 11.19 мы с помощью шаблона привязки данных добавляем текстовый элемент для каждого узла в наборе данных. Если метка связана с листовым узлом, то она отображается справа. В противном случае метка располагается слева от своего узла. Мы также добавляем белую обводку к меткам, чтобы сделать их более удобочитаемыми, когда они располагаются над связями. Устанавливая атрибут `paint-order` в значение `"stroke"`, мы гарантируем, что обводка будет иметь цвет заливки текста. Это также помогает с улучшением удобочитаемости.

Листинг 11.19 Добавление меток с названиями узлов (tree.js)

```
export const drawTree = (root, descendants) => {

  ...
  svg
    .selectAll(".label-tree")
    .data(descendants)
    .join("text")
      .attr("class", "label-tree")
      .attr("x", d => d.children ? d.y - 8 : d.y + 8)
      .attr("y", d => d.x)
      .attr("text-anchor", d => d.children ?
        "end" : "start")
      .attr("alignment-baseline", "middle")
```

Используя шаблон привязки данных, добавить текстовый элемент для каждого узла в наборе данных

Если метка соответствует листовому узлу, то расположить ее справа. В противном случае расположить ее слева от узла. Как упоминалось выше, значения `x` и `y` меняются местами, потому что дерево рисуется слева направо

```
    .attr("paint-order", "stroke")
    .attr("stroke", d => d.depth ===3 ?
      "none" : "white")
    .attr("stroke-width", 2)
    .style("font-size", "16px")
    .text(d => d.id);

};
```

Добавить к тексту белую обводку и установить для ее атрибута `paint-order` значение `"stroke"`, чтобы улучшить удобочитаемость

По завершении древовидная диаграмма должна выглядеть, как показано на рис. 11.13 или в онлайн-версии проекта (http://mng.bz/QZye).

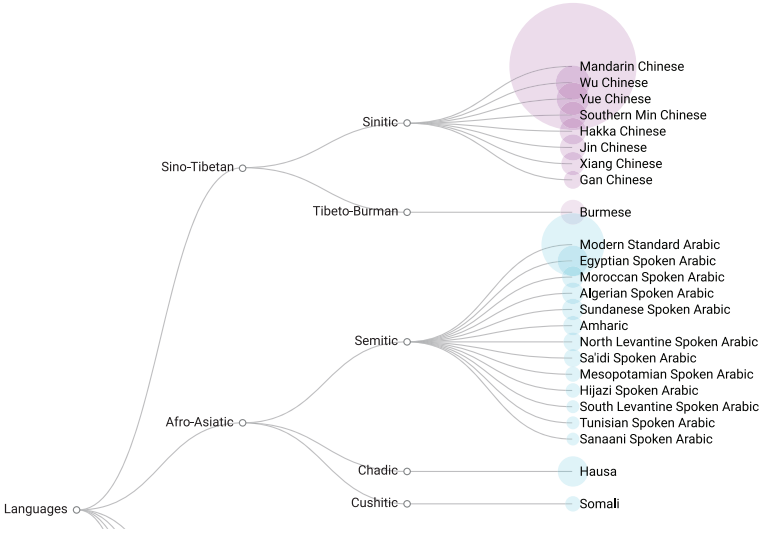


Рис. 11.13 Часть завершенной древовидной диаграммы

Линейная компоновка этой древовидной диаграммы позволяет относительно легко адаптировать ее для отображения на экранах мобильных устройств. Для этого достаточно увеличить размер шрифта меток и обеспечить достаточное вертикальное пространство между ними. Советы по созданию адаптивных диаграмм вы найдете в главе 9.

Чтобы, наконец, завершить этот проект, нужно добавить легенду, описывающую цвета семейств языков и размеры кругов. Мы уже создали ее. Чтобы отобразить легенду, перейдите в `index.html` и раскомментируйте `div` с классом `"legend"`. Затем в `main.js` импортируйте функцию `createLegend()` из `legend.js` и вызовите ее. Эта легенда создавалась по образу и подобию легенды к визуализации китообразных в главе 7, поэтому если что-то вам покажется непонятным, то обращайтесь к той главе. Внимательно изучите код в `legend.js` или, что еще лучше, попробуйте построить легенду самостоятельно!

11.4 Создание других иерархических визуализаций

В этой главе мы обсудили, как построить упакованную пузырьковую и древовидную диаграммы. Другие иерархические представления, такие как древовидные карты и сосульчатые диаграммы, создаются аналогично.

На рис. 11.14 показано, как, начиная с преобразования данных CSV или JSON в иерархическую структуру с именем root, можно по-

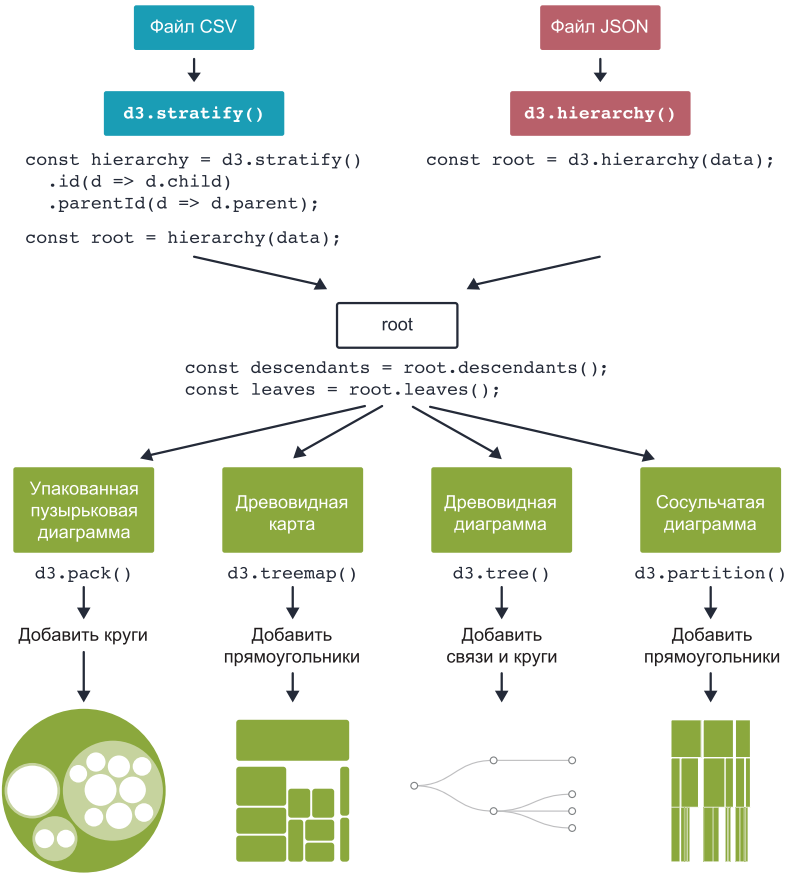


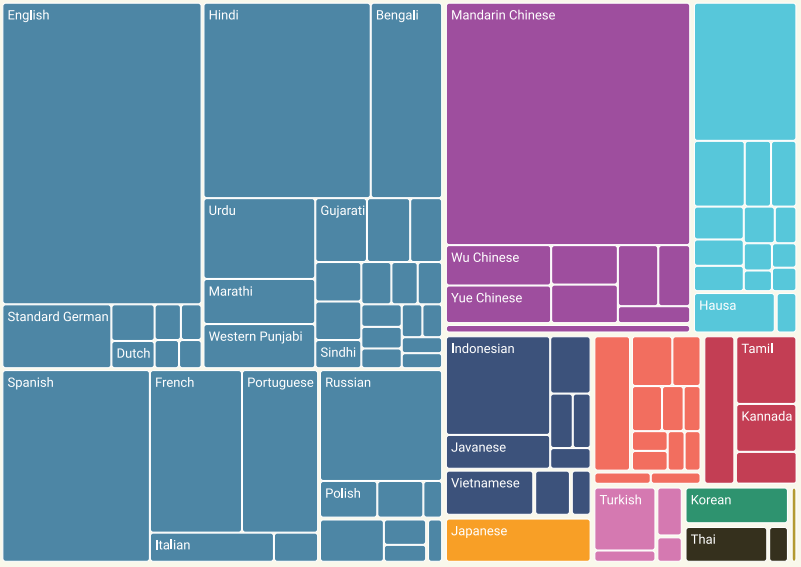
Рис. 11.14 Чтобы создать иерархическую визуализацию, сначала нужно преобразовать данные CSV или JSON в иерархическую структуру (обычно с именем root). На основе этой структуры можно сгенерировать упакованную пузырьковую диаграмму, древовидную карту, древовидную диаграмму или сосульчатую диаграмму. Для каждой диаграммы имеется свой генератор макета, добавляющий информацию, необходимую для построения диаграммы. Далее сгенерированная информация используется для рисования нужных нам фигур: кругов, прямоугольников или связей

строить любую иерархическую визуализацию. Единственное, что их отличает, – это функция, используемая в качестве генератора макета. Для упакованных пузырьковых диаграмм используется функция `d3.pack()`; для древовидных карт – `d3.treemap()`; для древовидных диаграмм – `d3.tree()`; а для сосульчатых диаграмм – `d3.partition()`. Все эти функции находятся в модуле d3-hierarchy.

Упражнение: создание древовидной карты

Теперь вы знаете все, что нужно, чтобы построить древовидную карту 100 самых распространенных языков, которую можно видеть на рисунке ниже и в онлайн-версии проекта (http://mng.bz/QZye).Древовидные карты визуализируют иерархические данные как набор вложенных прямоугольников. Традиционные древовидные карты показывают только конечные узлы, которые в нашем случае являются языками. Размер прямоугольников или конечных узлов пропорционален общему числу владеющих каждым языком. Вот несколько рекомендаций, которые помогут вам:

1. В `index.html` добавьте `div` с идентификатором `"treemap"`.
2. Создайте новый файл с именем `treemap.js` и с функцией `drawTreemap()`. Эта функция должна принимать структуру данных `root` и массив `leaves` и вызываться из `main.js`.
3. Используйте генератор макета `d3.Treemap()` для вычисления макета древовидной карты. С помощью функции доступа `size()` задайте ширину и высоту диаграммы. Также с помощью `paddingInner()` и `paddingOuter()` можно указать отступы между прямоугольниками. Более подробную документацию см. в модуле d3-hierarchy.



Древовидная карта 100 самых распространенных языков

4. Добавьте контейнер SVG в `div` с идентификатором `"treemap"`.
5. Добавьте прямоугольник для каждого листового узла. Установите его координаты и размер с помощью информации, добавленной генератором макета.
6. Добавьте метку для каждого языка в соответствующий прямоугольник. Метки для небольших прямоугольников можно скрыть.

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в разделе D.7 и в папке `11.4-Treemap/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

Интервью с Ширли Ву

Давая это интервью, Ву попросила, чтобы в качестве ее профессии было указано «просто человек».

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

После окончания университета я поступила на работу в компанию, где мы создавали информационные панели с помощью таких инструментов, как Highcharts, но компания была заинтересована в расширении круга возможностей. Примерно через год после выхода D3 мой тогдашний руководитель предложил изучить D3 и использовать ее для создания информационных панелей.

Изучая D3, я посетила несколько встреч в Сан-Франциско, где люди, хорошо знающие эту библиотеку, объяснили мне аспекты, с которыми у меня возникали трудности, например с шаблоном enter-update-exit.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашем рабочем процессе визуализации данных?

При создании цифровых визуализаций, помимо D3, я использую также Vue.js и GreenSock. Поскольку Vue.js отдает приоритет анимации, я обнаружила, что этот инструмент отлично работает с D3. GreenSock также позволяет создавать анимации, намного более сложные, чем переходы D3.

В последнее время я начала создавать статические визуализации, например для *Scientific American*. С этой целью я создаю графику SVG, созданную в D3, и экспортирую ее в Adobe Illustrator.

Представленный проект: «~~wonder~~ & hope» (www.wonder-and-hope.art/)



«До пандемии я любила гулять. Мне нравилось смотреть на ярко-голубое небо, чувствовать кожей дуновения ветра на коже и тепло солнечных лучей. Я смотрела, как распускаются цветы, слушала щебетание птиц, и меня охватывало чувство восторга» (фото предоставлено Туаном Хуаном (Tuan Huang), https://tuan-h.com/**)**

Расскажите о своем проекте «~~wonder~~ & hope».

Я очень люблю гулять и делаю много фотографий на прогулках. Во время пандемии у меня появился страх – я боялась выходить на улицу в одиночку, в основном из-за событий, связанных с ненавистью к азиатам. То, что раньше было источником восторга (прогулки на свежем воздухе), перестало быть таковым.

После вакцинации я постепенно начала выходить на улицу и поняла, что мне нужно восстанавливать свое отношение к пребыванию на улице. Я искала способ отразить свой опыт через фотографии, сделанные в годы до, во время и после пандемии. Я использовала метаданные фотографий (данные и местоположение), а также сценарий для извлечения основного цвета из каждой фотографии (я написала этот сценарий для проекта «Four Years in Vacations in 20,000 Colors», который создан в рамках проекта Data Sketches [https://shirleywu.studio/travel/]). Я использовала эти данные для создания визуализации с помощью D3.

Затем я построила различные физические слои для печати различных элементов визуализации: подложки, кругов, представляющих фотографии, меток и т. д. Они использовались, чтобы показать посетителям, как читать визуализацию.



«Первое требование оставаться дома было оглашено в Сан-Франциско 16 марта и действовало до мая. [...] В те же месяцы выходит статья с отчетом о преступлениях на почве ненависти к азиатам. Мы не выходим на улицу неделями, а в те редкие моменты, когда нужно выйти по делам, мы выходим группами» (фото предоставлено Туаном Хуаном (Tuan Huang), https://tuan-h.com/**)**

Визуализация также проецируется на воду, в которую падают капли, представляющие количество людей, умерших во время пандемии. В будущем они будут представлять число жертв преступлений, совершенных во время пандемии против азиатов на почве ненависти.

Наконец, в начале выставки представлены реальные цветы, олицетворяющие каждую фотографию, сделанную мной до пандемии, и один цветок в конце выставки, соответствующий одной конкретной фотографии цветов, сделанной мной после того, как я снова стала выходить из дома в одиночку.

Создавая цифровые проекты, я знаю, сколько людей посетили страницу, но прямой контакт с пользователями отсутствует. Создав такую физическую выставку, я могла видеть, как люди рассматривают ее, и они могли рассказать мне, чем их тронула инсталляция. Есть что-то очень важное, отличающее просмотр статьи на телефоне и посещение конкретной выставки, чтобы увидеть проект.

Что бы вы посоветовали начинающим специалистам по визуализации данных, которые никак не решатся «занять место» в этой области?

Интересный вопрос, потому что сейчас я совершаю поворот в своей карьере и направляюсь в искусство – новую для меня область. Я снова чувствую, что значит начинать карьеру заново и быть новичком. Но

каждый раз, когда я выкладываю свою очередную работу, я получаю обратную связь от людей, продвинувшихся дальше меня на этом пути. Эта обратная связь помогает мне совершенствоваться. И даже когда нет никакой реакции в сети, я по крайней мере документирую свою работу, и в этом процессе есть своя ценность.

Итоги

- Иерархические визуализации передают отношения родитель–потомок посредством *вложения*, *связей* или *смежности*.
- Их можно создавать из данных CSV или JSON.
- При работе с данными CSV нужны два столбца: один для дочерних узлов и один для их родителей. После загрузки набора данных в проект вызывается метод `d3.stratify()`, генерирующий иерархическую структуру данных, которой обычно дается имя `root`.
- При работе с иерархическими данными JSON используется метод `d3.hierarchy()`, генерирующий иерархическую структуру данных, которой обычно дается имя `root`.
- Иерархическая структура данных `root` имеет метод `descendants()`, возвращающий массив всех узлов, и метод `leaves()`, возвращающий массив всех листовых узлов (узлов, не имеющих потомков).
- Для создания упакованной пузырьковой диаграммы иерархическая структура `root` передается в генератор макета `d3.pack()`, который вычисляет координаты и размер каждого круга и добавляет эту информацию в набор данных. Затем ее можно использовать для рисования кругов, составляющих визуализацию.
- Для создания древовидной диаграммы иерархическая структура `root` передается в генератор макета `d3.tree()`, который вычисляет координаты каждого узла и добавляет эту информацию в набор данных. Затем ее можно использовать для рисования связей и кругов, составляющих визуализацию.

12 Сетевые визуализации

В этой главе рассматриваются:

- манипулирование узлами и связями в наборах данных;
- построение матрицы смежности;
- создание дуговой диаграммы;
- запуск симуляции с использованием макета d3-force;
- определение местоположений узлов путем балансировки сил притяжения и отталкивания, действующих между ними.

Сетевой анализ и сетевые визуализации стали обыденными в эпоху социальных сетей и больших данных. Они интересны тем, что фокусируются на связях между объектами и представляют системы более точно, чем традиционные плоские визуализации данных. В иерархических визуализациях (обсуждаемых в главе 11) узел может иметь много потомков, но только одного родителя. В сетевых же визуализациях есть возможность отразить связи «многие ко многим».

В общем случае при работе с сетями связанные объекты и субъекты (например, люди) называются *узлами*, а связи между ними (например, следование за кем-то в X, ранее Twitter) – *ребрами* или *связями*. Сети также могут называть *графами*, потому что так их называют в математике. Они могут представлять множество различных структур данных, таких как транспортные сети и связанные открытые данные.

Сети – это не просто формат данных: это перспектива данных. Работая с сетевыми данными, мы обычно пытаемся обнаружить и отобразить закономерности в сети или в ее части, а не отдельные узлы. В целом типичные методы визуализации информации предназначены для демонстрации структуры сети, а не отдельных узлов.

В этой главе мы рассмотрим четыре распространенные формы визуализации сетей: матрицы смежности, дуговые диаграммы, графики пчелиных роев и сетевые графики, изображенные на рис. 12.1, причем последние две будут созданы с применением силовой компоновки.

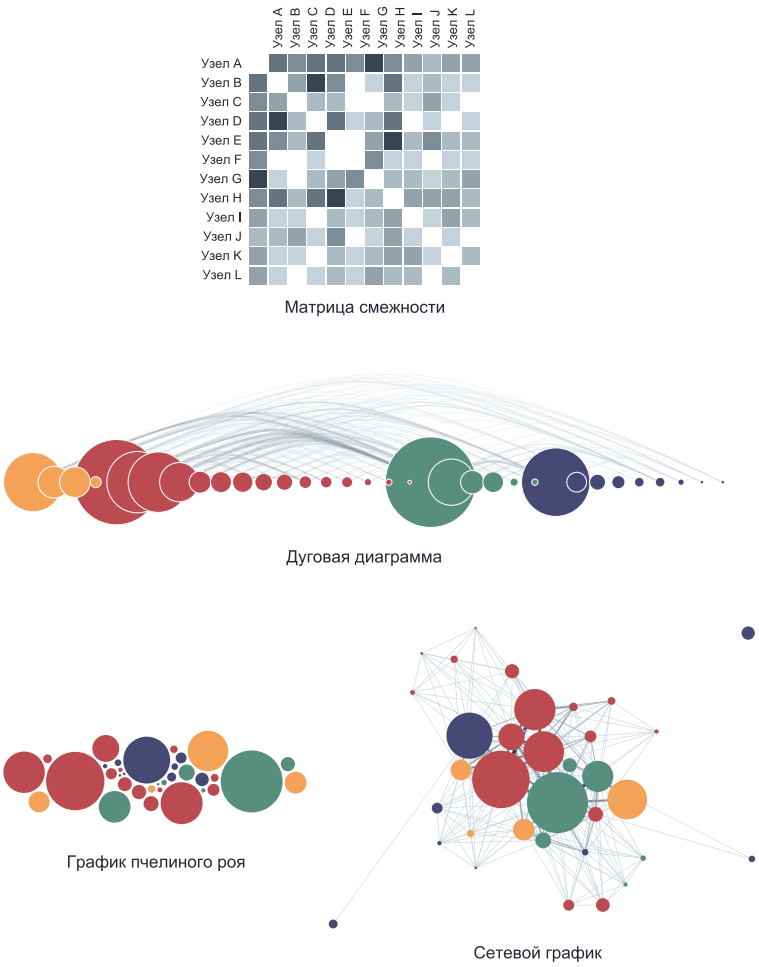


Рис. 12.1 Матрицы смежности, дуговые диаграммы, графики пчелиного роя и сетевые графы – четыре распространенных способа визуализации сетевых данных. Графики пчелиного роя фокусируются на узлах, а матрицы смежности используют сетку для представления связей между узлами. Дуговые диаграммы и сетевые графы отображают как узлы, так и связи между ними

12.1 Подготовка сетевых данных

В этой главе мы создадим проект на основе данных из пьесы Шекспира «Ромео и Джульетта». Онлайн-версия проекта доступна по адресу http://mng.bz/y8yE, а файлы с исходным кодом вы найдете в репозитории книги на GitHub (http://mng.bz/Xqjv).

Основной набор данных находится в файле `romeo_and_juliet.json`. В этом файле каждый объект – это реплика из пьесы, сопровождаемая именем персонажа, произносящего ее, а также номером акта и сцены. В табл. 12.1 показана часть этого набора данных из акта 1, сцен 3 и 5.

Таблица 12.1 Примеры реплик из пьесы «Ромео и Джульетта»

Акт, сцена, реплика	Персонаж	Текст реплики
1.3.1	Сеньора Капулетти	Где дочь моя? Пошли ее ко мне, кормилица!
1.3.2	Кормилица	Невинностью моей в двенадцать лет – клянусь,
1.3.3	Кормилица	уж я давно звала ее. Ягненочек мой, птичка!
1.3.4.	Кормилица	Куда ж она девалась? А? Джульетта!
1.3.5	Джульетта Капулетти	Что там? Кто звал меня?
1.5.115	Ромео Монтекки	Верни ж мой грех.
1.5.116	Джульетта Капулетти	Вина с тебя снята.
1.5.117	Кормилица	Синьора, ваша матушка вас просит.
1.5.118	Ромео Монтекки	Кто мать ее?

Для построения сетевых визуализаций нам нужно создать два отдельных файла из исходного набора данных: один с узлами и другой с ребрами. В нашем проекте узлами будут персонажи из пьесы. Всякий раз, когда два персонажа участвуют хотя бы в одной сцене, между ними образуется связь (ребро). Вес этой связи равен количеству сцен, в которых они участвуют вместе.

Если взять за основу пример набора данных из табл. 12.1, то файл данных узлов может выглядеть, как показано в табл. 12.2. Он содержит информацию о каждом узле. В нашем случае это имена персонажей, дома, к которым они принадлежат, их описание и количество произносимых ими реплик.

Таблица 12.2 Список узлов для образца набора данных

Персонаж	Дом	Описание	Количество реплик
Сеньора Капулетти	Дом Капулетти	Жена главы дома Капулетти	115
Кормилица	Дом Капулетти	Личная служанка и наперсница Джульетты	281
Джульетта Капулетти	Дом Капулетти	13-летняя дочь Капулетти, главная героиня пьесы	544
Ромео Монтекки	Дом Монтекки	Сын Монтекки, главный герой пьесы	614

Полный список узлов уже доступен в файле с именем `nodes.json`. Мы используем эту информацию при построении нашего проекта.

Второй файл – это список ребер. Список ребер имеет два основных столбца с именами *source (источник)* и *target (адресат)*. Как следует из названий, они представляют начало и конец каждой связи. На рис. 12.2 показаны два типа сетевых ребер: ориентированные и неориентированные. *Ориентированные ребра* направлены от узла-источника к узлу-адресату. Это направление может быть представлено как стрелка. Например, вы можете подписаться на получение новостей о каком-то пользователе в социальной сети, но тот пользователь может быть не подписан на получение новостей о вас. *Неориентированные ребра* представляют двунаправленную связь. В нашем проекте «Ромео и Джульетта» ребра будут неориентированными.

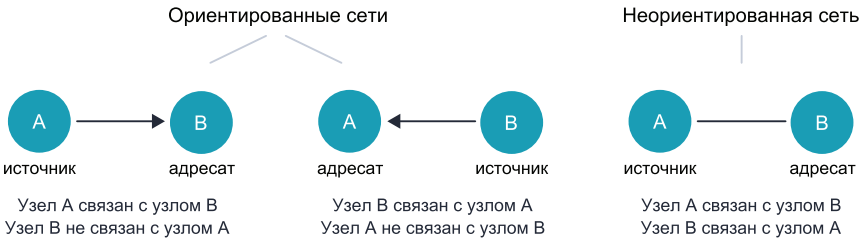


Рис. 12.2 Сетевые ребра имеют источник и адресата. В ориентированных сетях направление связи имеет определенный смысл и часто обозначается стрелкой. В неориентированных сетях связи не имеют определенных направлений

В табл. 12.3 показан пример, как будет выглядеть список ребер для нашего образца набора данных. В этом небольшом примере сеньора Капулетти и кормилица участвуют вместе только в одной сцене (1.3), тогда как кормилица и Джульетта участвуют вместе в двух сценах (1.3 и 1.5). Поскольку ребра в нашем примере ненаправленные, то неважно, какой персонаж является источником или адресатом, важно лишь их присутствие в одной строке в таблице.

Таблица 12.3 Список связей для образца набора данных

Источник	Адресат	Вес
Сеньора Капулетти	Кормилица	1
Кормилица	Джульетта Капулетти	2
Ромео Монтекки	Джульетта Капулетти	1

Чтобы указать количество сцен, в которых вместе участвуют персонажи, мы используем атрибут веса *weight*. Позднее этот вес будет влиять на близость расположения двух узлов в сетевом графе.

Полный список ребер уже доступен в файле с именем `edges.json`. Он содержит каждую пару персонажей, которые появляются вместе хотя бы в одной сцене. Атрибут веса указывает количество сцен, в которых они участвуют вместе. В файле `edges.json` это число представляет общее количество сцен, в которых два персонажа участвуют

вместе на протяжении всей пьесы. Мы используем эту информацию при построении наших визуализаций в следующих разделах.

12.2 Создание матрицы смежности

Вас может удивить, что одна из самых эффективных сетевых визуализаций вообще не имеет соединительных линий – для представления связей между узлами матрица смежности использует сетку. Принцип построения матрицы смежности прост: сначала узлы размещаются вдоль оси x, а затем вдоль оси y. Если узел на оси x связан с узлом на оси y, то соответствующий квадрат сетки закрашивается. В противном случае он остается пустым, как показано на рис. 12.3. Поскольку наша матрица имеет веса связей, то для их обозначения можно использовать насыщенность или непрозрачность, чтобы более светлые тона соответствовали более слабой связи, а более темные – более сильной.

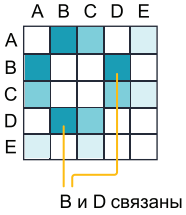


Рис. 12.3 В матрице смежности узлы размещаются сначала вдоль оси x, а затем снова вдоль оси y. Если два узла связаны, то соответствующий квадрат сетки закрашивается. Силу этой связи можно передать с помощью насыщенности цвета или непрозрачности

На рис. 12.4 показана матрица смежности, которую мы построим в этом разделе. Обратите внимание на симметричность матрицы. Такая симметрия объясняется неориентированностью сетевых связей. Если Ромео участвует в одной сцене с Джульеттой, то и Джульетта участвует в одной сцене с Ромео! Поскольку персонажи не могут быть связаны сами с собой, диагональные квадраты матрицы остаются пустыми.

Строятся матрицы смежности с помощью D3.js относительно просто, потому что эта визуализация состоит только из прямоугольников и текстовых элементов. В этом разделе мы посмотрим, как создать эту диаграмму на основе сетевых данных (файлов с узлами и связями). Мы также сделаем ее интерактивной, чтобы улучшить читаемость.

Для начала откройте в редакторе папку `start`, находящуюся в каталоге `chapter_12/12.2-Adjacency_matrix`. Этот проект использует модули JavaScript ES6, и мы уже добавили в него соответствующие зависимости D3.

Их можно увидеть в `package.json`. Чтобы установить зависимости, выполните команду `npm install` в терминале. Как объяснялось

в главе 11, чтобы запустить проект локально, нам понадобится упаковщик. Если вы уже установили Parcel, то можете сразу ввести команду `parcel src/index.html` в терминале, после чего проект станет доступен по адресу http://localhost:1234/. Если нет, то вернитесь к началу главы 11 и установите Parcel, следуя инструкциям, перечисленным там.



Рис. 12.4 Матрица смежности в проекте «Ромео и Джульетта». Персонажи (узлы) перечислены вдоль оси x и оси y, а количество сцен, в которых они участвуют вместе, сообщается цветом соответствующих квадратов сетки. Чем темнее квадрат, тем в большем количестве сцен персонажи участвуют вместе. Матрицы смежности неориентированных сетей симметричны, а матрицы смежности ориентированных сетей асимметричны

Мы уже загрузили содержимое файлов `nodes.json` и `edges.json` в проект. В `main.js` эти данные доступны в виде констант `nodes` и `edges`. Оставаясь в `main.js`, раскомментируйте следующий вызов функции:

```
drawMatrix(nodes, edges);
```

Затем откройте файл `matrix.js`, где объявлена эта функция. Поскольку в D3 нет генератора макета для матриц смежности, нам придется выполнить ряд вычислений самостоятельно. В листинге 12.1 мы сначала сортируем узлы в порядке убывания количества реплик. В этом порядке мы будем перечислять их в нашей матрице.

Массив ребер `edges` состоит из неориентированных связей. И для каждой пары персонажей, участвующих вместе хотя бы в одной сцене, имеется только одна связь. Но в матрицах смежности связи представлены в двух направлениях: от узлов, перечисленных вдоль оси `x`, к узлам, перечисленным вдоль оси `y`, и наоборот. Чтобы извлечь все требуемые связи, мы выполняем цикл по содержимому массива `edges` и для каждого ребра создаем две ссылки: одну от источника к адресату и одну от адресата к источнику. Информацию об этих ребрах мы сохраняем в объекте с именем `edgeHash` с ключами, соответствующими именам источника и адресата.

Наконец, мы создаем все возможные связи, дважды выполняя обход узлов. При этом мы вычисляем координаты верхнего левого угла соответствующего квадрата сетки и сохраняем их в атрибутах `x` и `y`. Таким способом мы создаем свой собственный макет. Затем, если `edgeHash` содержит соответствующую ссылку, мы помещаем элемент в массив `matrix`. По завершении массив `matrix` будет содержать по одному элементу для каждого квадрата в нашей визуализации.

Листинг 12.1 Подготовка матрицы данных (matrix.js)

```
export const drawMatrix = (nodes, edges) => {

  nodes.sort((a, b) => b.totalLinesNumber -
    a.totalLinesNumber);

  const edgeHash = {};
  edges.forEach(edge => {

    const link1 = {
      source: edge.source,
      target: edge.target,
      weight: edge.weight
    };
    const id1 = `${edge.source}-${edge.target}`;
    edgeHash[id1] = link1;

    const link2 = {
      source: edge.target,
      target: edge.source,
      weight: edge.weight
    };
    const id2 = `${edge.target}-${edge.source}`;
    edgeHash[id2] = link2;

  });
```

Отсортировать узлы в порядке убывания количества реплик

Извлечь все существующие связи (в обоих направлениях) и сохранить их в виде объекта в массиве с именем edgeHash. Этот объект поможет нам проверить, есть ли ссылка у пары источник–адресат

```
  const matrix = [];
  const squareWidth = 16;
  const padding = 2;
  nodes.forEach((charA, i) => {
    nodes.forEach((charB, j) => {
      if (charA !== charB) {
        const id = `${charA.id}-${charB.id}`;
        const item = {
          id: id,
          source: charA.id,
          target: charB.id,
          x: i * (squareWidth + padding),
          y: j * (squareWidth + padding)
        };

        if (edgeHash[id]) {
          item.weight = edgeHash[id].weight;
          matrix.push(item);
        }
      }
    });
  });

};
```

Создать все возможные связи источник–адресат, дважды обойдя узлы

Если в списке ребер есть соответствующее ребро, то присвоить элементу соответствующий вес и поместить его в массив matrix

Установить координаты x и y на основе индексов в массивах источников и адресатов. Мы используем эту информацию позже при размещении квадратов сетки

Упражнение: рисование матрицы смежности

Теперь, реализовав предварительную обработку данных, у вас есть все, что нужно, чтобы нарисовать матрицу смежности, подобную той, что можно увидеть на рис. 12.4 и в онлайн-версии проекта (http://mng.bz/y8yE). Вот некоторые рекомендации, которые помогут вам:

1. В `matrix.js` объявите функцию `drawMatrix()`.
2. Рассчитайте размеры диаграммы, исходя из количества узлов, размеров и отступов каждого квадрата.
3. Добавьте элемент SVG в `div` с идентификатором `matrix`, который уже существует в `index.html`.
4. Добавьте элемент прямоугольника для каждого элемента в массиве `matrix`. Используйте атрибуты `x` и `y`, чтобы задать местоположение квадратов. Присвойте их атрибутам `fill` цвет `"#364652"`, а атрибутам `fill-opacity` – значение, прямо пропорциональное весу ссылки (чем больше вес, тем выше уровень непрозрачности).
5. Для каждого узла добавьте метку слева от матрицы.
6. Для каждого узла добавьте метку сверху от матрицы. Используйте атрибут `transform` для поворота текстовых элементов.

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в разделе D.8 и в папке `12.1-Adjacency_matrix/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

По завершении матрица смежности должна выглядеть примерно так, как показано в онлайн-версии проекта. Чтобы помочь пользователю интерпретировать уровень непрозрачности квадратов, создадим легенду. Как показано в листинге 12.2, для этого сначала объявляется массив с именем `weights`, содержащий список целых чисел от 1 до 7. Они соответствуют возможным весам (количеству сцен, в которых вместе участвуют два персонажа).

Этот массив используется для создания неупорядоченного списка возможных весов. Каждый элемент списка содержит два элемента `div`, один отвечает за отображение цвета, другой – за отображение меток. Стили для этой легенды уже определены в `visualization.css`. Обязательно используйте те же имена классов, что и во фрагменте кода, чтобы задействовать эти стили.

Листинг 12.2 Добавление легенды для описания непрозрачности квадратов (matrix.js)

```
import { select, selectAll } from "d3-selection";
import { max, range } from "d3-array";
import { scaleLinear } from "d3-scale";

export const drawMatrix = (nodes, edges) => {

  ...

  const weights = range(1, maxWeight + 1);

  const legend = select(".matrix-legend")
    .append("ul")
    .selectAll(".legend-color-item")
    .data(weights)
    .join("li")
      .attr("class", "legend-color-item");
  legend
    .append("div")
      .attr("class", "legend-color")
      .style("background-color", "#364652")
      .style("opacity", d => opacityScale(d));
  legend
    .append("div")
      .attr("class", "legend-color-label")
      .text(d => d);

});
```

Создать массив, содержащий все возможные веса

Добавить неупорядоченный список в div с классом "matrix-legend". Использовать шаблон привязки данных, чтобы добавить в список элемент для каждого веса в массиве weights

Внутрь каждого элемента списка добавить div, отвечающий за отображение непрозрачности, соответствующей весу

Внутрь каждого элемента списка добавить div, отвечающий за отображение количества сцен, соответствующих каждому весу

Матрицы смежности может быть трудно читать, не имея возможности выделить строку и столбец, которым принадлежит некоторый квадрат. Добавив такую интерактивность, как показано в листинге 12.3, мы сделаем нашу визуализацию более удобной для чтения.

Листинг 12.3 Выделение связанных узлов при наведении указателя мыши (matrix.js)

```
...
import { transition } from "d3-transition";

export const drawMatrix = (nodes, edges) => {

  ...

  selectAll(".grid-quare")
    .on("mouseenter", (e, d) => {

      const t = transition()
        .duration(150);

      selectAll(".label-left")
        .transition(t)
          .style("opacity", label => label.id ===
            d.source ? 1 : 0.1);

      selectAll(".label-top")
        .transition(t)
          .style("opacity", label => label.id ===
            d.target ? 1 : 0.1);

      const charA = nodes.find(char => char.id ===
        d.source).name;
      const charB = nodes.find(char => char.id ===
        d.target).name;
      select(".matrix-tooltip-charA").text(charA);
      select(".matrix-tooltip-charB").text(charB);
      select(".matrix-tooltip-scenes").text(d.weight);
      select(".matrix-tooltip").classed("hidden", false);

    })
    .on("mouseleave", (e, d) => {

      selectAll(".label-top, .label-left")
        .style("opacity", 1);

      select(".matrix-tooltip").classed("hidden", true);

    });

});
```

Добавить обработчик события наведения указателя мыши на квадрат в матрице смежности

Установить уровень непрозрачности всех меток узлов персонажей равным 10 %, за исключением тех, что соответствуют источнику и адресату ребра

Отобразить подсказку с количеством сцен, в которых вместе участвуют два персонажа

Добавить обработчик события покидания указателем мыши квадрата в матрице смежности

Установить непрозрачность всех меток узлов персонажей равной 100 %

Скрыть подсказку

На рис. 12.5 можно видеть, как при наведении указателя мыши на квадрат в матрице выделяются соответствующие узлы и отображается подсказка с количеством сцен, в которых два персонажа участвуют вместе.

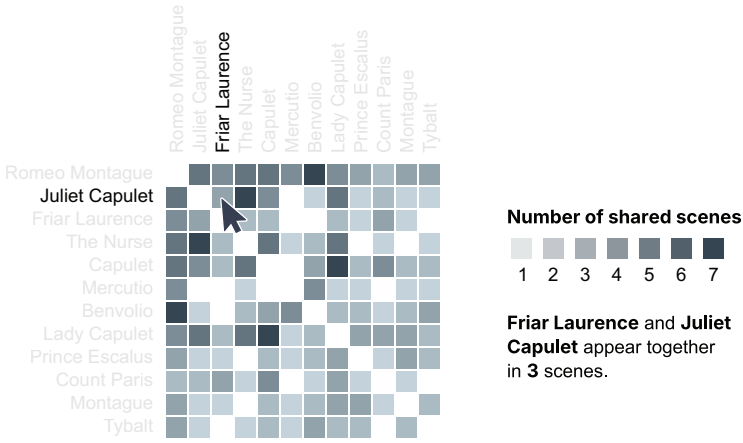


Рис. 12.5 При наведении указателя мыши на квадрат в матрице смежности выделяются метки узлов соответствующих персонажей и отображается подсказка с количеством сцен, в которых два персонажа участвуют вместе (здесь показана только часть диаграммы)

12.3 Создание дуговой диаграммы

Другой способ графического представления сетей предлагают дуговые диаграммы. На диаграммах этого вида узлы располагаются вдоль горизонтальной линии и затем выше или ниже рисуются связи между узлами в виде дуг. В отличие от матриц смежности, фокусирующихся на динамике ребер, дуговые диаграммы концентрируются на динамике узлов, позволяя видеть, какие узлы изолированы, а какие имеют много связей. На рис. 12.6 можно видеть, что узел A связан с узлами B и C, узел B – с узлами A и C, узел C – с узлами A и B, а узел D изолирован.

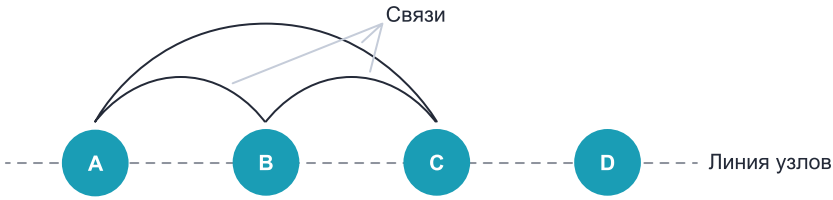


Рис. 12.6 На дуговых диаграммах узлы располагаются вдоль горизонтальной линии и затем выше или ниже рисуются связи между узлами в виде дуг. В этом примере узел A связан с узлами B и C, узел B – с узлами A и C, узел C – с узлами A и B, а узел D изолирован

В этом разделе мы построим дуговую диаграмму, показанную на рис. 12.7. Узлы персонажей представлены кругами и расположены

вдоль горизонтальной оси. Цвет узлов представляет семью, к которой они принадлежат (желтый соответствует Правящему дому Вероны, красный – дому Капулетти, зеленый – дому Монтекки, и фиолетовый – другим персонажам). Размер кругов пропорционален количеству реплик, которые они произносят на протяжении всей пьесы. Связь проводится между персонажами, участвующими вместе хотя бы в одной сцене. Чем больше количество общих сцен, тем толще связь.

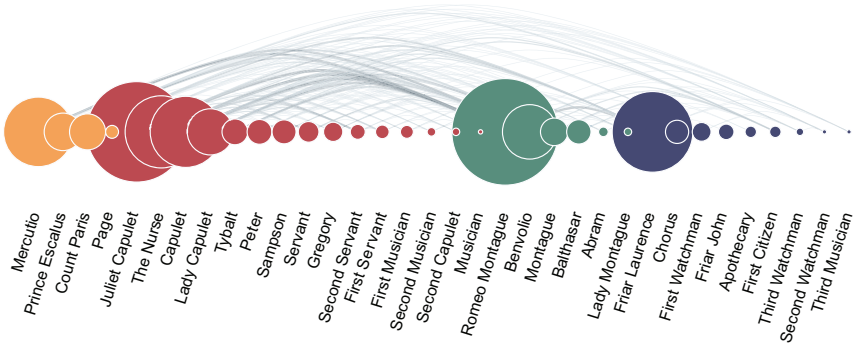


Рис. 12.7 Дуговая диаграмма, показывающая связи между персонажами из пьесы «Ромео и Джульетта». Чем толще связь, тем в большем количестве сцен участвуют два персонажа вместе

Итак, откройте файл `main.js` и раскомментируйте вызов функции `drawArcDiagram(nodes, edges)`. Затем откройте `arc.js`, в котором мы заполняли функцию `drawArcDiagram()`. Мы уже объявили размеры диаграммы.

Как и при создании матрицы смежности, мы должны предварительно обработать данные узлов и ребер, прежде чем рисовать диаграмму. В листинге 12.4 мы создаем полную копию массивов `nodes` и `edges`. Мы добавим в эти копии информацию, необходимую для построения дуговой диаграммы.

Построим псевдомакет, определив горизонтальное местоположение каждого узла с помощью точечной шкалы. Также исправим `edges` так, чтобы атрибуты источника и адресата ребра ссылались на объект узла, а не на его идентификатор. Позже это упростит доступ к начальной и конечной позициям ребер.

Листинг 12.4 Предварительная обработка данных дуговой диаграммы (arc.js)

```
import { scalePoint } from "d3-scale";
import { houses } from "./helper";

export const drawArcDiagram = (nodes, edges) => {

  ...
```

```
  const nodeHash = {};
  const arcNodes = JSON.parse(JSON.stringify(nodes));
  arcNodes.sort((a, b) => houses.find(h =>
    h.house === a.house).order - houses.find(h =>
    h.house === b.house).order);

  const xScale = scalePoint()
    .domain(arcNodes.map(node => node.id))
    .range([0, innerWidth]);

  arcNodes.forEach((node, i) => {
    nodeHash[node.id] = node;
    node.x = xScale(node.id);
  });

  const arcEdges = JSON.parse(JSON.stringify(edges));
  arcEdges.forEach(edge => {
    edge.source = nodeHash[edge.source];
    edge.target = nodeHash[edge.target];
  });

};
```

Создать полную копию массива узлов

Отсортировать узлы в порядке, установленном в массиве houses (предопределен в helper.js)

Объявить точечную шкалу, отвечающую за расчет местоположений узлов на горизонтальной оси

Создать хеш, связывающий каждый узел с его идентификатором

Установить координату x каждого узла

Создать полную копию массива ребер

Заменить идентификатор узлов указателем на объект узла

После завершения этапа предварительной обработки данных можно добавить фигуры, составляющие дуговую диаграмму. Для рисования дуг используем элементы `path`. Как показано в листинге 12.5, мы передаем генератору линий D3 координаты узла-источника, узла-адресата и точки посередине между ними и получаем значение для атрибута `d` элемента `path`. Мы также устанавливаем атрибут `stroke-width` каждой дуги в соответствии с ее весом. Наконец, мы вызываем цветовую шкалу, уже объявленную в `scales.js`, чтобы задать цвет узлов.

Листинг 12.5 Добавление дуг и узлов (arc.js)

```
...
import { line, curveCardinal } from "d3-shape";
import { max } from "d3-array";
import { getRadius, colorScale } from "./scales";

export const drawArcDiagram = (nodes, edges) => {

  ...

  const svg = select("#arc")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`)
    .append("g")
      .attr("transform", `translate(${margin.left},
        ${margin.top})`);
```

Добавить контейнер SVG и внутреннюю диаграмму

```
  const getArc = d => {
    const arcGenerator = line().curve(curveCardinal);
    const midX = (d.source.x + d.target.x) / 2;
    const midY = -Math.abs((d.source.x -
      d.target.x) / 6);
    const path = arcGenerator([
      [d.source.x, 0],
      [midX, midY],
      [d.target.x, 0]
    ]);
    return path;
  };
```

Вычислить атрибут `d` для кривой, начинающейся в узле-источнике, проходящей через точку посередине над узлами и заканчивающейся в узле-адресате

```
  svg
    .selectAll(".arc-link")
    .data(arcEdges)
    .join("path")
      .attr("class", "arc-link")
      .attr("d", d => getArc(d))
      .attr("fill", "none")
      .attr("stroke", "#364652")
      .attr("stroke-width", d => d.weight)
      .attr("stroke-opacity", 0.1)
      .attr("stroke-linecap", "round");
```

Использовать шаблон привязки данных, чтобы добавить элемент path для каждого ребра. Вызвать функцию getArc(), чтобы задать атрибут `d`, и присвоить атрибуту stroke-width значение, соответствующее весу связи

```
  const maxLines = max(arcNodes, d =>
    d.totalLinesNumber);
  svg
    .selectAll(".arc-node")
    .data(arcNodes.sort((a, b) => b.totalLinesNumber -
      a.totalLinesNumber))
    .join("circle")
      .attr("class", "arc-node")
      .attr("cx", d => d.x)
      .attr("cy", 0)
      .attr("r", d => getRadius(maxLines,
        d.totalLinesNumber))
      .attr("fill", d => colorScale(d.house))
      .attr("stroke", d => "#FAFBFF")
      .attr("stroke-width", 2);
```

Использовать шаблон привязки данных (чтобы добавить элемент круга для каждого узла) отсортированных по убыванию количества произносимых реплик. Присвоить их атрибуту `fill` цвет дома, используя уже объявленную в scales.js цветовую шкалу

```
  svg
    .selectAll(".arc-label")
    .data(arcNodes)
    .join("text")
      .attr("class", "arc-label")
      .attr("text-anchor", "end")
      .attr("dominant-baseline", "middle")
      .attr("transform", d => `translate(${d.x}, 70),
        rotate(-70)`)
      .text(d => d.name)
      .style("font-size", "14px");
};
```

Использовать шаблон привязки данных, чтобы добавить текстовую метку для каждого узла. Повернуть текст, чтобы упростить чтение

После этого шага ваша дуговая диаграмма будет выглядеть так же, как диаграмма в онлайн-версии проекта (http://mng.bz/y8yE). В такой абстрактной диаграмме интерактивность, помогающая определить, кто с кем связан, приобретает особую важность. Например, когда мышь находится над узлом, можно показать, какие персонажи связаны с этим узлом и соответствующие ребра. Именно это мы реализуем в листинге 12.6.

Листинг 12.6 Добавление интерактивности (arc.js)

```
...
import { transition } from "d3-transition";

export const drawArcDiagram = (nodes, edges) => {

  ...

  selectAll(".arc-node")
    .on("mouseenter", (e, d) => {

      const t = transition()
        .duration(150);

      const isLinked = char => {
        return arcEdges.find(edge =>
          (edge.source.id === d.id && edge.target.id
            === char.id) || (edge.source.id
            === char.id && edge.target.id === d.id))
          ? true
          : false;
      };

      selectAll(".arc-link")
        .transition(t)
        .attr("stroke-opacity", link =>
          link.source.id === d.id || link.target.id
            === d.id ? 0.1 : 0);

      selectAll(".arc-node")
        .transition(t)
        .attr("fill-opacity", char => char.id === d.id
          || isLinked(char) ? 1 : 0 )
        .attr("stroke-opacity", char => char.id ===
          d.id || isLinked(char) ? 1 : 0 );

      selectAll(".arc-label")
        .transition(t)
        .style("opacity", char => char.id === d.id
          || isLinked(char) ? 1 : 0 )
        .style("font-weight", char => char.id === d.id
          ? 700
          : 400);
```

Добавить обработчик события наведения указателя мыши на узел

Выяснить, какой персонаж связан с узлом, вызвавшим событие

Показать только узлы и ребра, связанные с узлом, вызвавшим событие

```
    })
    .on("mouseleave", (e, d) => {

      selectAll(".arc-link")
        .attr("stroke-opacity", 0.1);

      selectAll(".arc-node")
        .attr("fill-opacity", 1)
        .attr("stroke-opacity", 1);

      selectAll(".arc-label")
        .style("opacity", 1)
        .style("font-weight", 400);

    });

});
```

Добавить обработчик события покидания указателя мыши пределов узла

Вернуть непрозрачность связей и узлов в исходное состояние

На рис. 12.8 показано, как взаимодействие раскрывает динамику узлов. Исходя из предоставленной информации, мы можем заключить, что Ромео участвует хотя бы в одной сцене почти с каждым персонажем пьесы, в то время как брат Джиованни участвует в одной сцене только с братом Лоренцо.

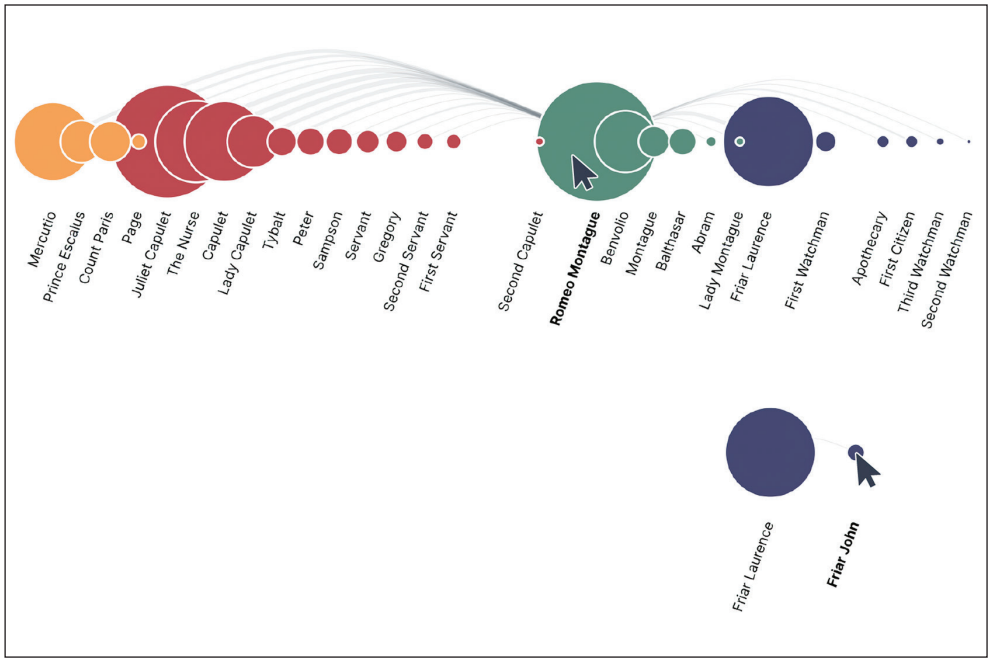


Рис. 12.8 Интерактивность помогает выявить динамику узлов. Мы видим, что Ромео участвует хотя бы в одной сцене почти с каждым персонажем пьесы, в то время как брат Джиованни участвует в одной сцене только с братом Лоренцо

12.4 Эксперименты с силами

Название «силовая компоновка» (force layout) происходит от метода, с помощью которого определяется наиболее оптимальное графическое представление сети. Он динамически обновляет позиции элементов, стремясь найти наилучшее соответствие. В отличие от макетов D3, обсуждавшихся ранее, силовая компоновка выполняет вычисления в реальном времени, а не во время этапа предварительной обработки перед отображением.

Принцип, лежащий в основе силовой компоновки, заключается в определении местоположений узлов путем балансировки сил притяжения и отталкивания, действующих между ними. Определяя силы притяжения, можно заставить узлы притягиваться к определенной позиции, а определяя силы отталкивания, можно гарантировать, что узлы не будут перекрываться. В этом разделе мы поэкспериментируем с этими различными силами и посмотрим, как они влияют на визуализацию. Мы также построим график пчелиного роя и сетевую диаграмму.

12.4.1 Запуск симуляции действующих сил

Запуск симуляции производится вызовом метода `d3.forceSimulation()` из модуля d3-force. На основе приложенных сил D3 добавляет в каждый узел два элемента информации: их следующее местоположение (`x` и `y`) и их следующую скорость (`vx` и `vy`). Каждый раз, когда D3 завершает итерацию симуляции, называемую тактом, она вызывает пользовательскую функцию, которая меняет местоположение узлов в соответствии с данными, добавленными библиотекой D3. Этот цикл (см. рис. 12.9) выполняется до тех пор, пока симуляция не достигнет оптимального состояния, соответствующего заданным силам.

Чтобы настроить первую нашу симуляцию, откройте `main.js` и раскомментируйте вызов функции `drawBeeswarm(nodes)`. Затем откройте файл `beeswarm.js` и найдите функцию `drawBeeswarm()`, где мы уже объявили размеры диаграммы. Далее мы продолжим наполнять эту функцию.

В листинге 12.7 мы добавляем контейнер SVG в DOM и элемент круга для каждого узла. Обратите внимание, что нам не нужно устанавливать атрибуты `cx` и `cy` кругов, это будет делаться во время симуляции.

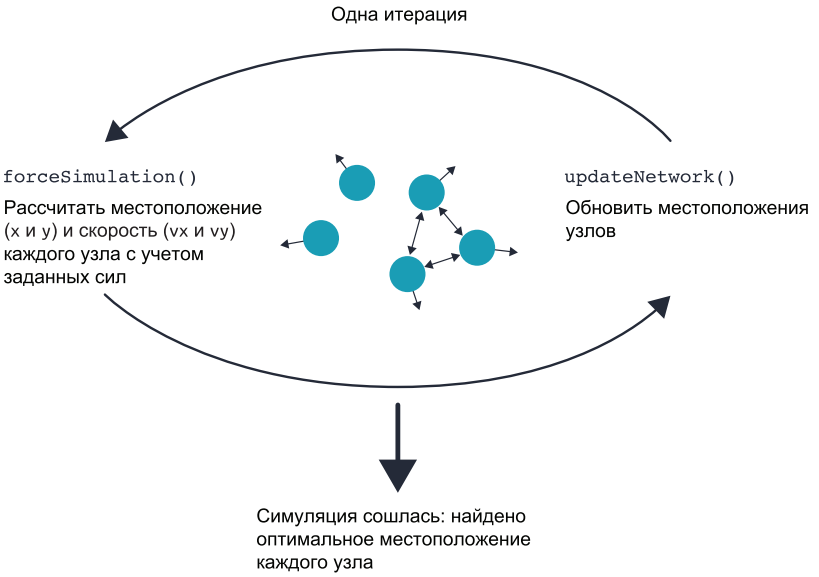


Рис. 12.9 При запуске симуляции силовой компоновки D3 выполняет последовательность итераций, в которых вычисляет местоположение и скорость каждого узла в соответствии с заданными силами. Затем вызывается пользовательская функция, которая обновляет местоположения узлов. Симуляция прекращается, как только будет найдено оптимальное местоположение для каждого узла

Далее объявим функцию с именем `updateNetwork()`, которая выбирает все узлы и устанавливает их координаты, исходя из атрибутов `x` и `y`, добавленных симуляцией. Чтобы запустить симуляцию, вызовем метод `forceSimulation()` и добавим в цепочку к нему вызов функции доступа `nodes()`, а также функцию обратного вызова `updateNetwork()`, которая будет запускаться после каждой итерации.

Листинг 12.7 Запуск симуляции (beeswarm.js)

```
import { select, selectAll } from "d3-selection";
import { forceSimulation } from "d3-force";
import { colorScale } from "./scales";

export const drawBeeswarm = (nodes) => {

  ...

  const svg = select("#beeswarm")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`)
    .append("g")
      .attr("transform", `translate(${width/2},
        ${height/2})`);
```

Добавить контейнер SVG и элемент группы для внутренней диаграммы. Переместить группу в центр контейнера SVG

```
  svg
    .selectAll(".beeswarm-circle")
    .data(nodes)
    .join("circle")
      .attr("class", "beeswarm-circle")
      .attr("r", 8)
      .attr("fill", d => colorScale(d.house))
      .attr("stroke", "#FAFBFF")
      .attr("stroke-width", 1);
```

Использовать шаблон привязки данных, чтобы добавить круг для каждого узла. Установить цвет кругов согласно принадлежности персонажей домам

```
  const updateNetwork = () => {
    selectAll(".beeswarm-circle")
      .attr("cx", d => d.x)
      .attr("cy", d => d.y);
  };
```

Объявить функцию, обновляющую местоположение каждого узла на основе атрибутов `x` и `y`

```
  const simulation = forceSimulation()
    .nodes(nodes)
    .on("tick", updateNetwork);

};
```

Вызвать d3.forceSimulation() для запуска симуляции. Добавить в цепочку функцию доступа к узлам и после каждой итерации вызывать функцию updateNetwork()

Если вы сохраните этот код и откроете проект в браузере, то увидите, что узлы аккуратно сгруппировались в середине контейнера SVG, как показано на рис. 12.10. С этой целью мы поместили внутреннюю диаграмму в центр контейнера SVG, как показано в листинге 12.7. В следующих подразделах мы добавим различные силы в симуляцию и посмотрим, как они влияют на местоположение узлов.



Рис. 12.10 Если в симуляции отсутствуют какие-либо силы, то узлы образуют кластер без перекрытия

Силы позиционирования

Первые силы, которые мы обсудим, – это силы позиционирования: `d3.forceX()` и `d3.forceY()`. Как следует из названий, `forceX()` заставляет узлы смещаться в направлении определенной координаты `x`, а `forceY()` – в направлении определенной координаты `y`. На рис. 12.11 показано, что добавление в симуляцию силы `forceX(0)` заставляет узлы занять местоположение с координатой `x` = 0 и позволяет свободно выбрать координату `y`. Сила `forceY(0)` действует ровно наоборот: заставляет узлы занять местоположение с координ-

натой y = 0 и позволяет свободно выбрать координату x. Если применить обе силы, `forceX(0)` и `forceY(0)`, то все узлы наложатся друг на друга в точке (0, 0).

```
forceSimulation()
  .force("x", d3.forceX(0) )
```



```
forceSimulation()
  .force("y", d3.forceY(0) )
```



```
forceSimulation()
  .force("x", d3.forceX(0) )
  .force("y", d3.forceY(0) )
```



Рис. 12.11 `d3.forceX()` заставляет узлы сместиться в определенное местоположение по горизонтали, а `d3.forceY()` – в определенное местоположение по вертикали

Величину действующей силы можно контролировать с помощью функции доступа `strength()`. По умолчанию, если не указано иное, величина действующих сил равна 1. Предположим, мы применяем сразу обе силы позиционирования, `forceX()` и `forceY(0)`, одновременно уменьшив величину силы `forceX()` до 0,01. В этом случае притяжение к координате (0,0) будет сильнее по вертикали, чем по горизонтали. На рис. 12.12 показано, что это приводит к тому, что узлы свободно выбирают местоположение на оси x. Если, наоборот, уменьшить величину силы `forceY()` до 0,01, притяжение узлов к координате (0,0) будет сильнее по горизонтали. Обратите внимание, что функцию доступа `strength` можно использовать с любой из сил, поддерживаемых библиотекой D3.

ПРИМЕЧАНИЕ Существует еще одна сила позиционирования, `d3.forceRadial()` (здесь не рассматривается), которую следует использовать для позиционирования узлов по окружности.

```
forceSimulation()
  .force("x", d3.forceX(0) )
  .force("y", d3.forceY(0) )
```

```
forceSimulation()
  .force("x", d3.forceX(0).strength(0.01) )
  .force("y", d3.forceY(0) )
```

```
forceSimulation()
  .force("x", d3.forceX(0) )
  .force("y", d3.forceY(0).strength(0.01) )
```



Рис. 12.12 С помощью функции `strength()` можно определять величину каждой действующей силы. По умолчанию эта величина равна 1. Если применить x-силу и y-силу, но ослабить x-силу, то притяжение к координате (0,0) будет сильнее по вертикали, чем по горизонтали. Если сделать наоборот, то притяжение к центру будет сильнее по горизонтали, чем по вертикали

Силы позиционирования можно также использовать для создания кластеров. Например, на рис. 12.13 демонстрируется пример использования точечной шкалы для определения горизонтального или вертикального положения узлов на основе дома персонажа. Передавая это значение силе позиционирования, мы создаем отдельные кластеры или группы для разных домов.

```
forceSimulation()
  .force("x", d3.forceX(d => xScale(d.house)) )
```

```
forceSimulation()
  .force("y", d3.forceY(d => yScale(d.house)) )
```

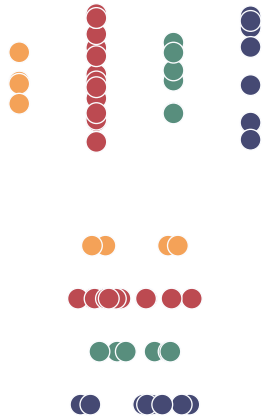


Рис. 12.13 Силы позиционирования можно использовать для создания кластеров. Передавая в `forceX()` или `forceY()` значение, возвращаемое точечной шкалой, можно создать отдельные группы для разных домов

Сила столкновения

Чтобы предотвратить перекрытие узлов, можем использовать силу столкновения `d3.forceCollide()`. Она принимает значение минимального расстояния между узлами. В примере, показанном на рис. 12.14, каждый узел имеет радиус 10 пикселей, а значение, переданное в функцию доступа `radius()`, равно `12`. Это означает, что сила столкновения не позволит приблизиться двум узлам друг к другу ближе, чем на 12 пикселей между их центрами, создавая отступ в 2 пикселя.

```
forceSimulation()
  .force("collide", d3.forceCollide().radius(12) )
```



```
forceSimulation()
  .force("x", d3.forceX(0) )
  .force("collide", d3.forceCollide().radius(12) )
```



```
forceSimulation()
  .force("y", d3.forceY(0) )
  .force("collide", d3.forceCollide().radius(12) )
```



Рис. 12.14 Сила столкновения `d3.forceCollide()` предотвращает перекрытие узлов. Добавляя в цепочку к ней функцию доступа `radius()`, можно задать минимальное расстояние между центрами узлов. В отсутствие сил позиционирования узлы естественным образом группируются в круглый кластер. Если в дополнение к силе столкновения применить силу позиционирования, то получится сплющенная форма, напоминающая пчелиный рой

В отсутствие сил позиционирования узлы естественным образом группируются в круглый кластер. Если в дополнение к силе столкновения применить `forceX()` или `forceY()`, то получится форма, похожая на пчелиный рой.

Давайте уделим немного времени изучению новой для нас возможности создания визуализации в виде пчелиного роя. Диаграммы пчелиного роя широко используются для отображения распределений с сохранением отдельных точек. Для целей демонстрации мы создадим 300 узлов и используем метод `d3.randomNormal()`, чтобы присвоить их атрибуту `value` случайное число, подчиняющееся нор-

мальному распределению, которое также называют распределением Гаусса:

```
const sampleArray = range(300);
sampleNodes = [];
sampleArray.forEach(() => {
  const randomNumberGenerator = randomNormal();
  sampleNodes.push({ value: randomNumberGenerator() * 10 });
});
```

Предположим, что мы добавляем эти узлы в контейнер SVG и запускаем симуляцию действия силы `forceX()`, чтобы расположить по горизонтали в соответствии со значением их атрибута `value` (установленного в предыдущем фрагменте кода). В этом случае сила `forceY()` притянет узлы к центру по вертикали, а сила столкновения гарантирует отсутствие перекрытия. В итоге мы получим диаграмму пчелиного роя:

```
forceSimulation()
  .force("x", forceX(d => xScale(d.value)) )
  .force("y", forceY(0) )
  .force("collide", forceCollide().radius(7) )
  .nodes(sampleNodes)
  .on("tick", updateNetwork);
```

Как показано на рис. 12.15, мы действительно получаем нормальное распределение. Горизонтальное положение каждого узла должно быть довольно близко к значению, определяемому его атрибутом `value`, тем не менее полезно знать, что D3 пытается найти положения, которые уравновешивают все силы, участвующие в симуляции. Окончательные горизонтальные положения могут не точно соответствовать атрибуту `value`.

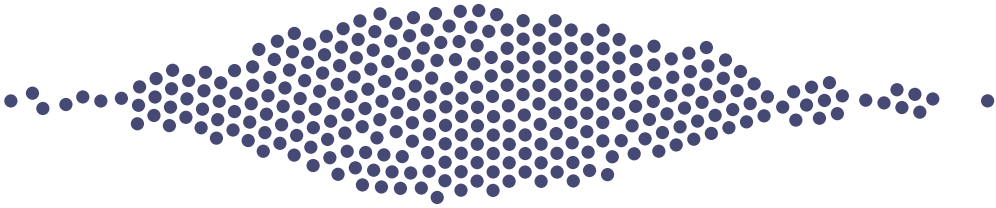


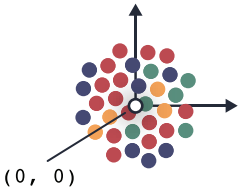
Рис. 12.15 Диаграмма пчелиного роя, представляющая нормальное распределение

Сила центрирования

Сила центрирования `d3.forceCenter()` перемещает весь кластер узлов в заданное местоположение. Если силы позиционирования по осям `x` и `y` перемещают частицы по отдельности и могут сплющить систему частиц, то центрирующая сила перемещает всю систему

частиц, сохраняя ее первоначальную форму. На рис. 12.16 к группе узлов сначала применяется сила столкновения. По умолчанию узлы стремятся расположиться вокруг координаты (0,0). Если применить силу центрирования с помощью `d3.forceCenter()` и посредством функций доступа `x()` и `y()` задать координаты для смещения, то все узлы вместе переместятся в указанные координаты.

```
forceSimulation()
  .force("collide", d3.forceCollide().radius(12) )
```




```
forceSimulation()
  .force("center", forceCenter().x(-10).y(5) )
  .force("collide", d3.forceCollide().radius(12) )
```

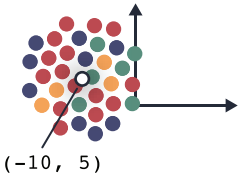


Рис. 12.16 Сила центрирования перемещает систему частиц или узлов целиком в направлении координат, переданных функциям доступа `x()` и `y()`

Многочастичная сила

До сих пор мы обсуждали только силы, притягивающие узлы к некоторой позиции. Многочастичная сила `d3.forceManyBody()` отличается тем, что описывает силу взаимодействия узлов друг с другом. Отрицательная величина многочастичной силы имитирует отталкивание, тогда как положительная величина имитирует притяжение, как показано на рис. 12.17.

Чтобы создать визуализацию персонажей из пьесы «Ромео и Джульетта» в стиле пчелиного роя, которую можно увидеть в онлайн-версии проекта (http://mng.bz/y8yE), масштабируем радиус узлов на основе реплик, произнесенных соответствующим персонажем. Для этого вызовем функцию `getRadius()`, уже имеющуюся в `scales.js`, а также сохраним радиус каждого узла в связанных данных. Мы воспользуемся этим значением чуть позже, чтобы задать силу отталкивания между узлами. Мы также создадим полную копию массива `nodes` и используем ее в симуляции. Это избавит нас от проблем при запуске двух силовых симуляций с использованием одних и тех же данных (пчелиный рой и сетевая диаграмма):

```
const beeswarmNodes = JSON.parse(JSON.stringify(nodes));

const maxLines = max(nodes, d => d.totalLinesNumber)
svg
```

```
.selectAll(".beeswarm-circle")
.data(beeswarmNodes)
.join("circle")
  .attr("class", "beeswarm-circle")
  .attr("r", d => {
    d.radius = getRadius(maxLines, d.totalLinesNumber);
    return d.radius;
  })
  .attr("fill", d => colorScale(d.house))
  .attr("stroke", "#FAFBFF")
  .attr("stroke-width", 1);
```

```
forceSimulation()
```

```
forceSimulation()
  .force("charge", forceManyBody().strength(-10))
```

```
forceSimulation()
  .force("charge", forceManyBody().strength(10))
```

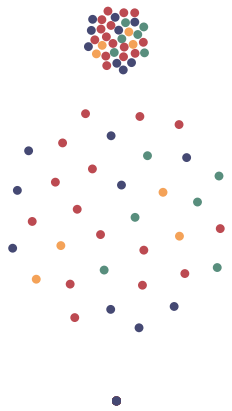


Рис. 12.17 Многочастичная сила определяет взаимодействия узлов друг с другом. Отрицательная величина многочастичной силы имитирует отталкивание, а положительная – притяжение

Затем применим силу позиционирования по оси y, чтобы притянуть узлы к центру контейнера SVG по вертикали. Мы также используем силу столкновения, чтобы исключить перекрытие кругов. Наконец, добавим функцию доступа `radius`, извлекающую значение атрибута `radius`, который мы добавили к узлам, и прибавляющую к этому значению 2 пикселя:

```
forceSimulation()
  .force("y", forceY(0) )
  .force("collide", forceCollide().radius(d => d.radius + 2) )
  .nodes(beeswarmNodes)
  .on("tick", updateNetwork);
```

Получившаяся диаграмма пчелиного роя показана на рис. 12.18.



Рис. 12.18 Диаграмма пчелиного роя, изображающая персонажей из пьесы «Ромео и Джульетта». Цвет кругов соответствует домам персонажей, а размер – количеству реплик

Связи

Последняя сила, доступная в модуле d3-force, – это сила связи `d3.forceLink()`, которая притягивает друг к другу взаимосвязанные узлы: чем сильнее связь, тем ближе друг к другу стремятся расположиться узлы. Объединив силу связи с другими силами, рассмотренными ранее, мы сможем создать сетевую диаграмму.

Продолжим работу с нашим набором данных «Ромео и Джульетта» и создадим сетевую диаграмму, где связи представляют количество сцен, в которых вместе участвуют два персонажа. Откройте файл `main.js` и раскомментируйте вызов функции `drawNetwork(nodes, edges)`. Затем откройте файл `network.js`, начнем работать внутри функции `drawNetwork()`.

В следующем фрагменте кода мы используем функцию доступа `id()`, чтобы сообщить D3, как распознавать узлы-источники и узлы-адресаты. Обратите внимание, что узлы должны иметь уникальные идентификаторы. Мы также используем функцию доступа `strength()`, чтобы указать, что сила связи между узлами пропорциональна значению атрибута `weight` связи (количество сцен, в которых два персонажа, объединяемых этой связью, участвуют вместе).

Код в листинге 12.8 напоминает код создания диаграммы пчелиного роя в предыдущем разделе. Главное отличие – помимо кругов мы добавляем в диаграмму элемент линии для каждого ребра. Затем после каждой итерации вызывается функция `updateNetwork()`, которая обновляет не только позиции узлов, но и позиции связей.

Мы объединяем силу связи с силой центрирования, которая перемещает визуализацию в центр контейнера SVG, сильной отрицательной многочастичной силой, чтобы создать эффект отталкивания между узлами и добавить некоторое расстояние между ними, и силой столкновения, чтобы избежать наложения.

Листинг 12.8 Создание сетевой диаграммы (network.js)

```
import { select, selectAll } from "d3-selection";
import { max } from "d3-array";
import { forceSimulation, forceCollide, forceCenter, forceManyBody,
         forceLink } from "d3-force";
import { colorScale, getRadius } from "./scales";

export const drawNetwork = (nodes, edges) => {

  const width = 850;
  const height = 600;

  const svg = select("#network")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`)
    .append("g")
```

```
      .attr("transform", `translate(${width/2}, ${height/2})`);

  svg
    .selectAll(".network-link")
    .data(edges)
    .join("line")
      .attr("class", "network-link")
      .attr("stroke", "#364652")
      .attr("stroke-opacity", 0.1)
      .attr("stroke-width", d => d.weight);
```

Использовать шаблон привязки данных, чтобы добавить элемент линии для каждого ребра. Местоположение линий пока не устанавливается

```
  const maxLines = max(nodes, d => d.totalLinesNumber)
  svg
    .selectAll(".network-node")
    .data(nodes)
    .join("circle")
      .attr("class", "network-node")
      .attr("r", d => {
        d.radius = getRadius(maxLines, d.totalLinesNumber);
        return d.radius;
      })
      .attr("fill", d => colorScale(d.house))
      .attr("stroke", "#FAFBFF")
      .attr("stroke-width", 1);

  const updateNetwork = () => {
    selectAll(".network-link")
      .attr("x1", d => d.source.x)
      .attr("y1", d => d.source.y)
      .attr("x2", d => d.target.x)
      .attr("y2", d => d.target.y);
    selectAll(".network-node")
      .attr("cx", d => d.x)
      .attr("cy", d => d.y);
  };
```

Обновить положение связей после каждой итерации

```
  const simulation = forceSimulation()
    .force("charge", forceManyBody()
      .strength(-1000))
    .force("collide", forceCollide()
      .radius(d => d.radius + 2) )
    .force("center", forceCenter().x(0).y(0))
    .force("link", forceLink()
      .id(d => d.id)
      .strength(d => d.weight/10))
    .nodes(nodes)
    .on("tick", updateNetwork);

  simulation
    .force("link")
    .links(edges);
};
```

Объединить силу связи с силой центрирования, которая перемещает визуализацию в центр контейнера SVG, отрицательной многочастичной силой для создания эффекта отталкивания между узлами, чтобы создать между ними некоторое расстояние, и силой столкновения, чтобы избежать перекрытия

Установить массив edges, определяющий силы связи

Нестандартная ограничивающая сила

Открыв получившуюся сетевую диаграмму в браузере, можно заметить, что в ней отсутствует узел, представляющий хор. Причина в том, что этот узел не имеет связей, соединяющих его с другими узлами (у хора есть свои сцены, но в этих сценах не участвуют другие персонажи), а многочастичная сила применяет сильное отталкивание. Если посмотреть, где находится узел хора в инспекторе, то вы увидите, что он оказался за пределами контейнера SVG. Чтобы вернуть этот узел обратно, мы создадим нестандартную ограничивающую силу. Поскольку силы в D3 – это просто функции обратного вызова, ничто не мешает нам написать свою собственную!

В следующем фрагменте показана наша функция силы с именем `bounding`. Она циклически перебирает все узлы, и если какой-то из них оказывается за пределами контейнера SVG, то она изменяет его скорость (`vx` или `vy`), так чтобы направить узел обратно в контейнер SVG:

```
const simulation = forceSimulation()
  ...
  .force("bounding", () => {
    nodes.forEach(node => {
      if (node.x < -width/2 + node.radius) {
        node.vx = 1;
      }
      if (node.y < -height/2 + node.radius) {
        node.vy = 1;
      }
      if (node.x > width/2 - node.radius) {
        node.vx = -1;
      }
      if (node.y > height/2 - node.radius) {
        node.vy = -1;
      }
    });
  })
```

На рис. 12.19 показана получившаяся сетевая диаграмма. Главные персонажи (крупные круги) находятся близко друг к другу, потому что они вместе участвуют в нескольких сценах. Второстепенные персонажи (меньшие круги) находятся на периферии, потому что связи, соединяющие их с другими, слабее.

Чтобы завершить этот проект, добавим интерактивности и будем выделять связи при наведении указателя мыши. Что-то похожее мы уже реализовали в дуговой диаграмме. Когда указатель мыши оказывается над узлом, непрозрачность всех узлов и связей будет устанавливаться равной 0, за исключением узлов, связанных с выбранным узлом. Соответствующий код показан в листинге 12.9. До-

полнительно мы отобразим имя и описание персонажа в подсказке справа от визуализации.



Рис. 12.19 Сетевая диаграмма персонажей из пьесы «Ромео и Джульетта». Круги представляют персонажей, а их размеры пропорциональны количеству реплик, произносимых ими в пьесе. Связи между узлами представляют количество сцен, в которых они участвуют вместе. Чем ближе два узла друг к другу, тем в большем количестве сцен они участвуют вместе

Листинг 12.9 Добавление интерактивности (network.js)

```
export const drawNetwork = (nodes, edges) => {

  ...

  selectAll(".network-node")
    .on("mouseenter", (e, d) => {

      const t = transition()
        .duration(150);
      const isLinked = char => {
        return edges.find(edge =>
          (edge.source.id === d.id && edge.target.id === char.id) ||
          (edge.source.id === char.id && edge.target.id === d.id))
          ? true
          : false;
      };

      selectAll(".network-link")
        .transition(t)
```

```
        .attr("stroke-opacity", link =>
          link.source.id === d.id || link.target.id === d.id ? 0.1 : 0);

      selectAll(".network-node")
        .transition(t)
        .attr("fill-opacity", char =>
          char.id === d.id || isLinked(char) ? 1 : 0 );

      select(".network-character")
        .text(d.name);
      select(".network-description")
        .text(d.description);
      select(".network-sidebar")
        .classed("hidden", false);
```

Отобразить имя и описание выбранного персонажа в подсказке справа. Необходимая разметка уже имеется в index.html, а стили определены в visualization.css

```
    })
    .on("mouseleave", () => {

      selectAll(".network-link")
        .attr("stroke-opacity", 0.1);

      selectAll(".network-node")
        .attr("fill-opacity", 1);

      select(".network-sidebar")
        .classed("hidden", true);
```

Скрыть подсказку

```
    });

});
```

При наведении указателя мыши на узел видимыми остаются только те персонажи, которые участвуют хотя бы в одной сцене с выбранным, как показано на рис. 12.20.

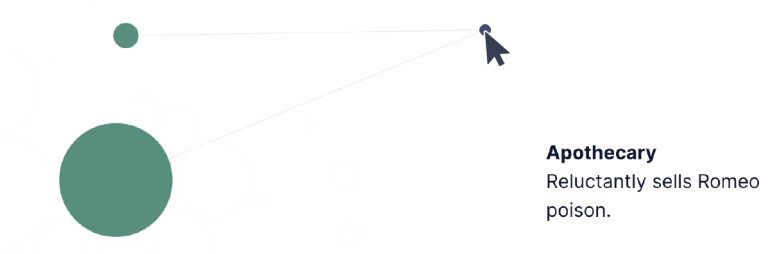


Рис. 12.20 Когда указатель мыши оказывается над узлом, то видимыми остаются только те персонажи, которые участвуют хотя бы в одной сцене с выбранным, а также соответствующие связи

Остановка и повторный запуск симуляции силовой компоновки

Симуляция силовой компоновки постепенно «замерзает» и останавливается по достижении некоторого баланса сил. После остановки симуляции иногда требуется возобновить анимационный эффект, вновь запустив анимацию. Кроме того, после изменения каких-либо настроек сил или добавления либо удаления узлов сети тоже бывает нужно остановить и повторно запустить симуляцию.

Остановить симуляцию можно вызовом `simulation.stop()`. Симуляцию желательно останавливать всегда, когда происходит взаимодействие с компонентами в другом месте на веб-странице или изменяются настройки визуализации, а затем повторно запускать ее по завершении взаимодействий.

Запустить или перезапустить симуляцию можно вызовом `simulation.restart()`. Явно запускать симуляцию при ее первом создании не требуется – она запустится автоматически.

Наконец, выполнить одну итерацию компоновки можно вызовом `simulation.tick()`. Кроме того, если вам не нужна забавная анимация на странице, то вы можете рассчитать диаграмму без симуляции. Например, с помощью вызова `simulation.tick(120)` можно выполнить 120 итераций, перед тем как отображать визуализацию. Симуляция без графической анимации происходит намного быстрее, и при желании вы можете использовать переходы D3 для анимации перемещения узлов в их окончательное местоположение по завершении расчетов.

Оптимизация

Силовая компоновка очень ресурсоемка. Вот почему предусмотрены ее постепенное «замерзание» и остановка. А если силовая компоновка применяется для отображения очень большой сети, то можно исчерпать вычислительные ресурсы компьютера пользователя и сделать его практически непригодным для использования. Первый совет по оптимизации – ограничьте количество узлов и ребер в вашей сети. Общепринятое правило: сеть не должна содержать больше 1000 узлов. Раньше этот предел был равен 100, но постепенно увеличивался и продолжает увеличиваться с ростом производительности браузеров, поэтому выявляйте экспериментальным путем, насколько большую сеть смогут отобразить браузеры, используемые вашей аудиторией.

Если нужно отобразить больше узлов и вы хотите повысить производительность, то используйте `forceManyBody.chargeDistance()`, чтобы задать максимальное расстояние, на котором будут вычисляться сила отталкивания для каждого узла. Чем меньше значение этого параметра, тем менее структурированной будет получаться силовая компоновка, но тем быстрее она будет работать. Поскольку сети сильно различаются, вам придется поэкспериментировать с разными значениями `chargeDistance()`, чтобы найти лучшее для вашей сети.

Наконец, если диаграмма не предполагает взаимодействий с пользователем, то подумайте о возможности делегирования вычислений веб-потоку или визуализации частей графика с помощью холста. О том, как использовать холст, мы поговорим в главе 15.

Интервью с Маттиасом Шталем

Шталь – дизайнер/разработчик визуализаций данных, в настоящее время занимает пост руководителя отдела визуализации данных в *DER SPIEGEL*.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

Страсть к визуализации данных поздно пришла ко мне. До 2020 года я занимался исследованиями в области естественных наук, так как изначально обучался по специальности «Биохимия». В аспирантуре я изучал бактерии, устойчивые к широкому кругу антибиотиков. Мы пытались выяснить на молекулярном уровне, почему эти паразиты становятся устойчивыми и как разрабатывать новые лекарства для борьбы с ними.

Мы провели множество экспериментов и получили огромные объемы данных. Как результат возникла необходимость проанализировать эти данные, что заставило меня изучить R и визуализацию данных. Позже я открыл для себя целый набор инструментов визуализации, позволяющих нам, исследователям, добиваться успеха: правильная визуализация часто оказывается ключом к новому прорыву.

В то же время я немного устал от исследований в лаборатории. Да, мы занимались важными медицинскими исследованиями, но они были слишком абстрактны и слишком далеки от реальной жизни людей. Постепенно я начал задумываться о том, чтобы заняться консалтингом по визуализации данных. Мне потребовался примерно год, чтобы укрепиться в своей мысли, и в конечном итоге я ушел из академии. Как показало время, это было верное решение. У меня появилась возможность поработать над захватывающими проектами разной тематики, не только научными. Кроме того, публикация визуализаций данных давала немедленный эффект: люди реагировали, комментировали и обсуждали мои работы. В конечном итоге мой интерес к данным и статистике встретился с захватывающими областями искусства и дизайна, а также психологии, так что я никогда не жалел, что выбрал этот путь.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашем рабочем процессе визуализации данных?

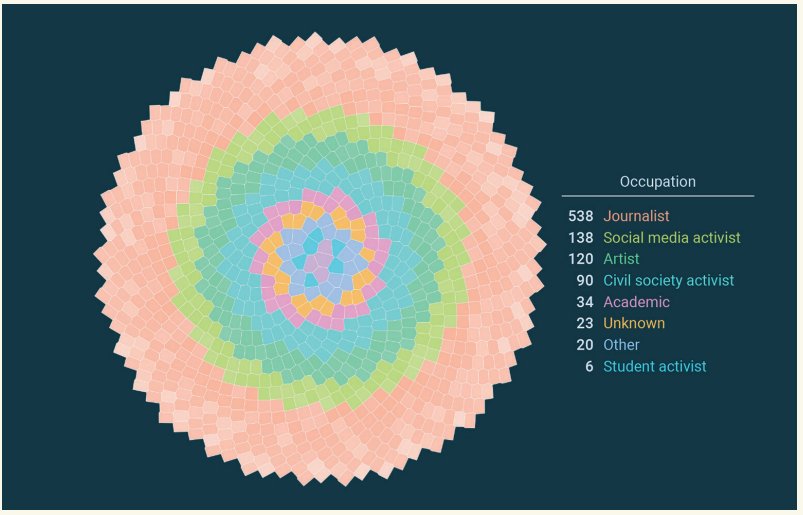
Практически все свои визуализации я создаю для публикации в интернете. Поэтому мои основные инструменты – это HTML, CSS и JavaScript. Конечно, писать программный код, используя только базовые возможности JavaScript, было бы слишком утомительно, поэтому я еще использую фреймворк Svelte, чтобы избавиться от шаблонного кода. Он также помогает мне разбивать визуализации на повторно используемые компоненты, что весьма важно в быстро меняющейся обстановке новостей, публикацией которых занимается наша редакция. В этих компонентах я использую D3 в качестве библиотеки вспомогательных функций. Как правило, я использую эти функции для вычисления значений атрибута `d` элементов `path`, для математических преобразований, таких как масштабирование или геопроекции, и для геометрических вычислений. Мой друг Коннор Ротшильд (Connor Rothschild) однажды сказал: «D3 хорошо использовать для работы с данными, а Svelte – с DOM», – и я согласен с ним. При этом я перестал использовать распространенный шаблон обновления в D3 и объединяю Svelte и D3 так, чтобы писать декларативный, а не императивный код. Например, я всегда настраиваю SVG как элементы HTML (`<svg>`) в Svelte, а затем применяю функции D3 для импорта данных.

Вы работаете в DER SPIEGEL. В чем уникальность вашего опыта создания визуализаций данных и проектов scrollytelling (последовательного повествования с помощью прокручиваемого контента) для новостного журнала?

Я работаю в Der SPIEGEL уже 2,5 года – сначала в качестве заместителя руководителя отдела графики, а затем в качестве руководителя недавно созданного отдела данных и визуализации. Мой опыт как дизайнера, разработчика и руководителя по визуализации данных имеет две отличительные черты. С одной стороны, у нас очень большая и заинтересованная аудитория, и мы обычно имеем полную свободу придумывать любые темы и идеи для понравившихся нам сюжетов. С другой стороны, в нашей среде сюжеты чаще всего связаны с текущими новостями. Поэтому наши проекты должны выполняться в очень сжатые сроки, что несколько ограничивает пространство для творчества. Последнее можно преодолеть с помощью тщательно разработанных фреймворков и библиотек. Например, для создания больших визуальных историй мы используем Svelte и D3, которые помогают создавать визуализации в кратчайшие сроки. Кроме того, мы собрали коллекцию компонентов и шаблонов Svelte, которые мы можем легко интегрировать в новые проекты.

В конце концов, именно мы формируем будущее визуализации и журналистики данных в редакциях. Мы, и наши коллеги в других редакциях, имеем свободу проверять инновационные идеи, и, что самое важное, наша аудитория также открыта для этих экспериментов.

Представленный проект: «Visualizing Censorship in Iran», созданный для организации «Journalism is Not a Crime» (https://journalismisnotacrime.com)



Этот проект визуализации представляет иранских журналистов и активистов, подвергшихся цензуре

Проект «Visualizing Censorship in Iran» – это визуализация, представляющая иранских журналистов и активистов, подвергшихся преследованиям со стороны правительства, задержаниям, тюремному заключению, изгнанию и в некоторых случаях убийству. Организация «Journalism Is not a Crime» документирует эти зверства и ведет подробную базу данных. Визуализация использует данные, собранные организацией, и рассказывает истории всех этих людей.

Расскажите, как создавался этот проект.

Когда я подключился к проекту, данные были готовы к визуализации – о таком начале можно только мечтать. Затем я приступил к исследованию дизайна и для проверки различных идей использовал только ручку и бумагу.

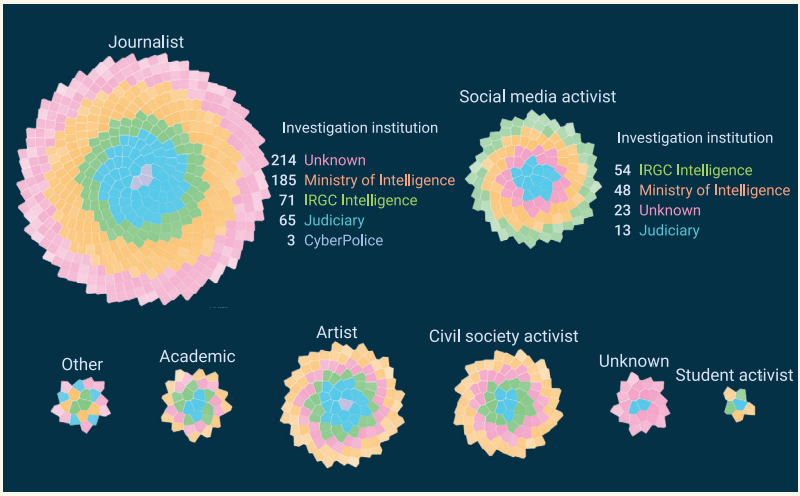
Параллельно я набирался впечатлений об Иране. В какой-то момент мое внимание привлекла персидская архитектура и ее красочные глазурованные мозаичные плитки. Перед нашим проектом стояла задача визуализировать огромное количество людей (около 700 на тот момент) и при этом выделить индивидуальные биографии. Использование метафоры плиток показалось мне хорошим решением, охватывающим оба требования: они уникальны сами по себе и производят разный эффект, когда рассматриваются в комбинации с другими плитками.

Идею плиток было трудно набросать от руки, так как требовалось сразу же использовать реальные точки данных. Поэтому я тут же приступил

к разработке, используя Svelte и D3, попутно черпая вдохновение из блокнотов Observable.

Мы хотели показать статистику и биографии, поэтому я разработал инструмент для получения различных точек зрения на полный набор данных и при этом позволяющий погрузиться в отдельные истории людей в данных. Быстро выяснилось, что анимация дает лучшее решение задачи связывания различных состояний и привлечения внимания читателей.

Через пару недель конечный продукт был готов к публикации. Его код можно найти на странице https://github.com/higsch/censorship-in-iran.



Каждая плитка представляет журналиста или активиста, которых пользователь может группировать, например, по учреждению, занимающемуся расследованием, роду занятий или полу

Дизайн на основе мозаичных плиток выглядит потрясающе! Трудно ли было сгенерировать их программно и отрисовать с помощью холста?

Спасибо за комплимент! Плитки действительно выглядят уникальными, и каждая имеет свою геометрию. Однако создать их было не так сложно, как вы могли подумать. Я сконструировал их, сначала отобразив множество точек в форме зубчатого колеса. Затем использовал центры точек для построения диаграммы Вороного. Последнюю легко построить с помощью библиотеки Delaunay в D3. Это все, что нужно для создания розетки из плиток.

Поскольку плитки могут находиться в разных розетках, их необходимо анимировать из одного состояния в другое. Для отображения более 700 плиток лучше всего использовать HTML-холст – SVG-графика была бы слишком медленной для этого. Рисование индивидуальных кривых на холсте легко может превратиться в мучение, но использо-

вание плиток Вороного, сгенерированных с помощью D3, в сочетании с компонентами Svelte (без фактического отображения) существенно упростило задачу. Вооружившись этой идеей (которая, кстати, существует и в других фреймворках пользовательского интерфейса), мы создали родительский компонент холста, реализующий рисование и принимающий плитки как дочерние элементы. Компоненты, такие как плитки, мы называем компонентами «без отображения», потому что сами они не отображают себя, а просто передают информацию о своем внешнем виде родительскому элементу, который отрисовывает их на холсте.

Как изменился ваш рабочий процесс по мере развития D3 с того момента, когда вы начали заниматься визуализацией данных в 2020–2021 годах, и до настоящего момента?

Видите ли, я использую D3 как библиотеку функций, откуда выбираю то, что мне нужно для решения конкретной задачи, поэтому мой рабочий процесс, как я описал его выше, не изменился. Но я по-прежнему не перестаю удивляться, когда время от времени нахожу функции, которых не замечал раньше.

Над какими проектами вы надеетесь работать в будущем?

Я мечтаю когда-нибудь создать инсталляцию с очень большим экраном или холстом в музее. Было бы заманчиво поработать над специально адаптированным дизайном и довести до совершенства взаимодействие с высоким разрешением.

Конечно же, для реализации такого проекта я бы использовал Svelte и D3.

Итоги

- Существует много способов представления сетей, например с помощью матриц смежности, дуговых диаграмм и силовых макетов. Всегда старайтесь выбирать метод, который лучше других подходит для отображения структуры вашей сети и вашей аудитории.
- Для построения сетевой визуализации необходимо создать два отдельных файла с данными из исходного набора: один с узлами и другой с ребрами.
- Матрицы смежности фокусируются на связях между узлами, тогда как дуговые диаграммы подчеркивают динамику узлов.
- Силовая компоновка `d3.forceSimulation()` динамически обновляет позиции элементов, стремясь найти их оптимальное расположение.
- Для управления силовой компоновкой используется комбинация сил притяжения и отталкивания (позиционирования, столкновения, центрирования, многочастичных и связи).

- Силы позиционирования притягивают узлы к определенным координатам. Их также можно использовать для создания кластеров узлов.
- Силы столкновения используются, чтобы установить минимальное расстояние между узлами и избежать перекрытия.
- Силы центрирования притягивают весь кластер к определенным координатам.
- Многочастичные силы влияют на взаимодействия узлов друг с другом. Отрицательная многочастичная сила имитирует отталкивание, а положительная – притяжение.
- Сила связи притягивает друг к другу узлы, соединенные связями. Чем больше вес связи, тем сильнее они будут притягиваться друг к другу под действием силы связи.
- Также есть возможность определять нестандартные силы, например не позволяющие узлам покидать пределы определенной области.
- Силовая компоновка требует больших ресурсов, поэтому для отображения большого количества узлов следует использовать методы оптимизации.

13 Визуализации геопространственной информации

В этой главе рассматриваются:

- манипулирование данными в форматах GeoJSON и TopoJSON;
- сравнение проекций и их влияние на макеты карт;
- нанесение точек на карту;
- включение функций масштабирования и панорамирования;
- обработка событий выделения мышью.

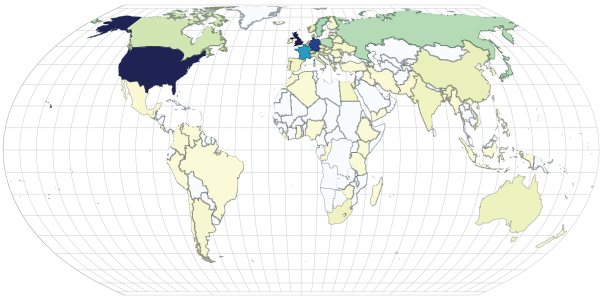
Последний тип визуализации, который мы обсудим в этой части, – карты. Карты представляют символическую интерпретацию геопространственных данных. Они используют различные проекции для изображения поверхности земного шара, на котором мы живем, на двумерной плоскости.

За многие годы существования картографирования, географических информационных систем (ГИС) и географической информатики было разработано множество методов представления такого рода данных. В D3 тоже имеются надежные функции для загрузки и отображения геопространственных данных, большинство из которых включены в модуль d3-geo.

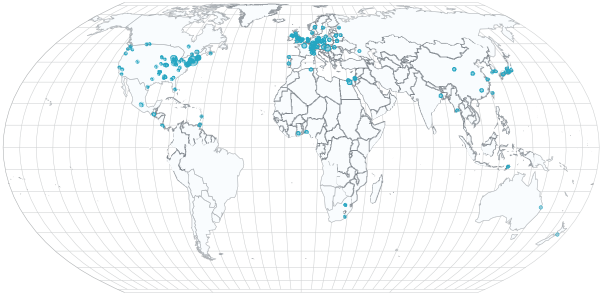
В этой главе мы используем два типа географических данных: GeoJSON и TopoJSON. Мы изучим различные проекции, из которых

будем создавать карты. Мы также добавим в свои визуализации стандартные функции, такие как масштабирование и панорамирование, и довольно редкие, такие как обработка события выделения объектов, отображаемых на карте.

При создании визуализаций мы будем использовать набор данных со списком лауреатов Нобелевской премии с 1901 по 2022 год и попытаемся ответить на следующий вопрос: «Как место вашего рождения влияет на ваши шансы получить Нобелевскую премию?» Мы построим карту мира, отметив на ней страны и города, где родились лауреаты Нобелевской премии, и более подробную карту Франции, как на рис. 13.1. Проект в законченном виде вы можете найти по адресу http://mng.bz/MZdm.



Карта мира в равновеликой проекции



Карта мира в равновеликой проекции
с нанесенными пропорциональными отметками



Карта Франции в проекции Меркатора
с нанесенными пропорциональными отметками

Рис. 13.1 В этой главе мы построим карту мира, используя равновеликую проекцию (проекция Равной Земли) и данные в формате GeoJSON, и карту Франции, используя проекцию Меркатора и данные в формате TopoJSON

13.1 Географические данные

Географические данные могут иметь вид простых координат (широта и долгота) или более сложных форм, таких как линии и другие геометрические фигуры, и поставляться в файлах форматов GeoJSON или TopoJSON. В этой главе мы рассмотрим оба этих формата.

13.1.1 GeoJSON

GeoJSON – это вариант представления геоданных в формате JSON. Данные в этом формате представляют определения точек, линий и полигонов и используются для рисования простых географических объектов. Каждый объект в файле GeoJSON должен иметь три свойства: `type`, `properties` и `geometry`. Свойство `type` может иметь одно из двух значений: `Feature` и `FeatureCollection`, где последнее означает, что объект представляет коллекцию.

Каждый объект содержит свойство `properties` с метаинформацией, такой как описательное имя объекта. Он также имеет свойство `geometry`, определяющее тип геометрической формы и ее координаты. В следующем фрагменте показан пример данных в формате GeoJSON, описывающих многоугольник, изображенный на рис. 13.2. Каждая пара координат (широта и долгота) представляет точку в декартовой системе координат:

```
{
  "type":"Feature",
  "geometry":{
    "type":"Polygon",
    "coordinates":[
      [
        [25,56],
        [23,55],
        [22,54],
        [26,54],
        [25,56]
      ]
    ]
  },
  "properties":{"name":"Country Name"},
  "id":"Country Code"
}
```

ПРИМЕЧАНИЕ Долготы (или меридианы) – это вертикальные линии, определяющие расстояние к востоку или западу от Гринвичского меридиана, а широты (или параллели) – это горизонтальные линии, определяющие расстояние к северу или югу от экватора.

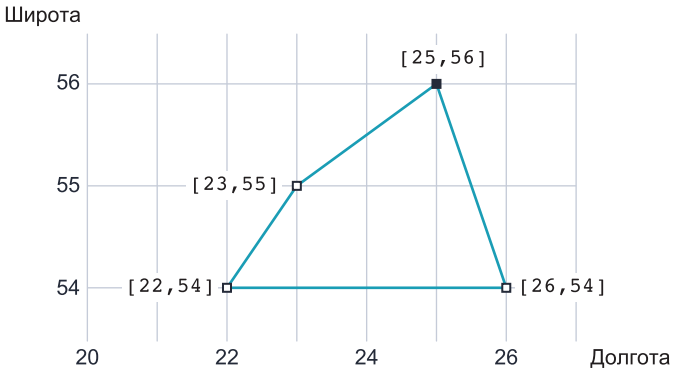


Рис. 13.2 Полигон и его координаты, указанные в файле GeoJSON

Если геометрическая форма представляет замкнутую фигуру, то начальная и конечная точки в списке имеют одинаковые координаты, как, например, пара координат `[25, 56]` в примере выше.

В файле GeoJSON, на основе которого мы будем строить карту мира (`world.json` в репозитории этой главы), каждый объект соответствует стране, а геометрические фигуры, описываемые ими, называются мультиполигонами. Но геометрические фигуры в GeoJSON также могут быть простыми точками и линиями.

Мы не будем вдаваться в детали определения геометрических фигур в формате GeoJSON, потому что вам едва ли когда-нибудь придется создавать свои собственные файлы в формате GeoJSON. Но если вы хотите погрузиться глубже, то загляните в статью с описанием GeoJSON в Википедии, где приводятся полезные примеры описания каждой геометрической формы (https://ru.wikipedia.org/wiki/GeoJSON).

Главное достоинство формата GeoJSON – простота. Но файлы в этом формате могут быть очень большими, поскольку он не оптимизирован для компактного представления информации и может хранить повторяющуюся информацию. Этот недостаток особенно ярко проявляется при картографировании больших территорий с большой детализацией. Когда требуется компактность представления, используется формат TopoJSON.

13.1.2 TopoJSON

В отличие от формата GeoJSON, который хранит массив координат для каждого объекта, описывающего точку, линию или полигон, TopoJSON хранит только массив дуг. Дуга – это любой отдельный сегмент линии, общий для одного или нескольких объектов в наборе данных.

Большинство наборов данных имеют общие сегменты, например границу между двумя странами. В GeoJSON общие границы описы-

ваются дважды, TopoJSON устраняет дублирование, объединяя их в одну дугу, как показано на рис. 13.3. В результате файлы получаются значительно меньше и лучше подходят для распространения через интернет и хранения сведений с большой детализацией. Кроме того, если известно, какие сегменты являются общими, то с данными можно производить разные интересные манипуляции, например быстро вычислить общую границу или объединение территорий.

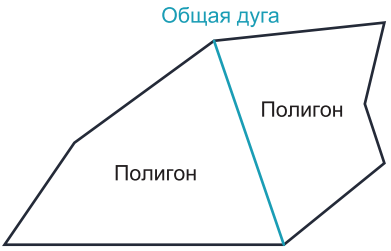


Рис. 13.3 TopoJSON устраняет дублирование, объединяя общие границы в одну дугу

Кроме того, в отличие от простого формата GeoJSON, который можно читать с помощью библиотеки D3, TopoJSON требует использования дополнительной библиотеки `topojson-client` для декодирования данных TopoJSON в формат GeoJSON, чтобы затем прочитать их с помощью D3.

Поиск географических данных

Простого поиска в Google часто бывает достаточно, чтобы найти файлы данных, необходимые для создания карт. Тем не менее от себя можем порекомендовать обратить внимание на практичный блокнот Observable, созданный Деей Банковой (Dea Bankova), графическим журналистом из Reuters, где можно найти полезные фрагменты кода D3 и ссылки на различные ресурсы (http://mng.bz/aERY).

13.2 Рисование карты на основе данных GeoJSON

В этом разделе мы нарисуем карту мира на основе данных GeoJSON. Затем отметим на этой карте страны и города, откуда родом лауреаты Нобелевской премии.

13.2.1 Выбор проекции

О создании веб-карт написаны целые книги, и об использовании D3.js для рисования карт тоже можно написать целую книгу. Но так

как нам доступна лишь одна глава, мы опустим обсуждение многих проблем. Одна из них – выбор проекции. В ГИС под проекцией понимается процесс отображения на плоскости точек на поверхности Земли. Проекции неизбежно искажают истинные формы и размеры географических объектов.

D3 поддерживает четыре распространенных типа проекций: азимутальную, коническую, цилиндрическую и составную. *Азимутальные проекции*, такие как равновеликая проекция «Равной Земли», показанная на рис. 13.4, проецируют сферу непосредственно на плоскость. Центр проекции – это точка касания плоскости с поверхностью Земли, и искажение увеличивается по мере удаления от этой центральной точки.

Конические проекции проецируют поверхность Земли на конус, который затем разворачивается в плоскость. Они лучше всего подходят для картографирования больших расстояний с востока на запад, потому что искажения вдоль параллелей постоянны.

Цилиндрические проекции проецируют поверхность Земли на цилиндр, который затем разворачивается в плоскость. В результате страны, расположенные ближе к полюсам, визуально выглядят больше, чем есть на самом деле. Проекция Меркатора, показанная на рис. 13.4, является типичным примером цилиндрической проекции. Обратите внимание, насколько преувеличенно большими выглядят Гренландия и Антарктида. Эта проекция использовалась в ранних версиях Google Maps и используется по умолчанию в Tableau и Power BI.

Наконец, *составные проекции* формируются из нескольких проекций. Они особенно полезны для показа отдаленных регионов стран, например Аляски и Гавайев вместе с континентальной частью США на одном листе. Обратите внимание на эффект проекции на этом примере. Несмотря на то что Аляска больше Техаса в два с лишним раза, она кажется меньше.

Создание карты с помощью D3 производится в три шага:

1. выбор проекции;
2. инициализация функции-генератора элементов `path`;
3. добавление элементов `path` в контейнер SVG.

Следующий фрагмент иллюстрирует реализацию двух первых шагов. Здесь мы сначала объявляем тип проекции и сохраняем его в константе с именем `projection`. В этом примере использована проекция `d3.geoEqualEarth()`, но точно так же можно было бы использовать любую другую проекцию из числа доступных в модуле d3-geo. Обратите внимание, что мы связываем ее с функциями доступа `transition()` и `scale()`, которые используются для масштабирования карты и ее центрирования внутри контейнера SVG.



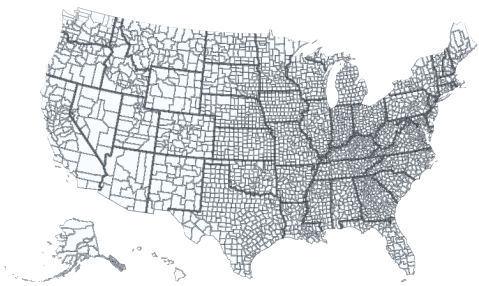
Естественная проекция Земли
(азимутальная)



Равновеликая коническая проекция Альберса
(коническая)



Проекция Меркатора (цилиндрическая)



Составная проекция США

Рис. 13.4 D3 поддерживает четыре распространенных типа проекций: азимутальную, коническую, цилиндрическую и составную. На этом рисунке показан пример каждой из них

Далее, вызовом `d3.geoPath()`, мы инициализируем функцию-генератор для создания элементов `path` и передаем проекцию, определенную ранее, в функцию доступа `projection()`. Этот генератор действует подобно генераторам форм, представленным в главе 5: ему передается объект в формате GeoJSON, а он возвращает значение для атрибута `d` элемента `path`:

```
const projection = d3.geoEqualEarth()
  .translate([translationX, translationY])
  .scale(factor);

const geoPathGenerator = d3.geoPath()
  .projection(projection);
```

Теперь нарисуем карту мира, как показано на рис. 13.1 и в онлайн-версии проекта (http://mng.bz/MZdm). Для начала откройте в редакторе папку `start`, находящуюся в каталоге `chapter_13/13.2-`

GeoJSON. Эту папку можно найти в репозитории книги GitHub по адресу http://mng.bz/Xqjv.

Данный проект создан с использованием модулей JavaScript ES6, и мы уже добавили в него необходимые зависимости D3. Вы можете увидеть их в файле package.json. Чтобы установить эти зависимости, выполните команду npm install в терминале. Как объяснялось в главе 11, чтобы запустить проект локально, вам понадобится упаковщик. Если вы уже установили Parcel, то можете сразу ввести команду parcel src/index.html в терминале, после чего проект станет доступен по адресу http://localhost:1234/. Если нет, то вернитесь к началу главы 11 и установите Parcel, следуя инструкциям, перечисленным там.

Папка data проекта содержит три файла:

- laureates.json – содержит сведения о лауреатах Нобелевской премии, такие как страна и город рождения;
- world.json – набор геоданных в формате GeoJSON;
- france.json – набор геоданных в формате TopoJSON для Франции.

Эти файлы уже загружаются в main.js, где также вызывается функция drawWorldMap(), которой передается набор данных laureates и геоданные в формате GeoJSON. Для начала откройте файл map-world.js и найдите функцию drawWorldMap().

Как показано в листинге 13.1, функция начинается с объявления размеров карты и добавления контейнера SVG в div с атрибутом id = map, который уже существует в index.html. Затем вызовом d3.geoEqualEarth() объявляется равновеликая проекция и сохраняется в константе с именем projection. Обратите внимание, что пока мы не использовали ее функции доступа scale() и translate(). Мы поэкспериментируем с ними чуть ниже.

Далее объявляется функция-генератор элементов path, для чего вызывается метод d3.geoPath(), которому через функцию доступа projection() передается проекция. В заключение мы используем шаблон привязки данных и добавляем элемент path для каждого объекта, присутствующего в world.features, вызывая функцию-генератор geoPathGenerator для установки атрибута d.

Листинг 13.1 Рисование карты мира (map-world.js)

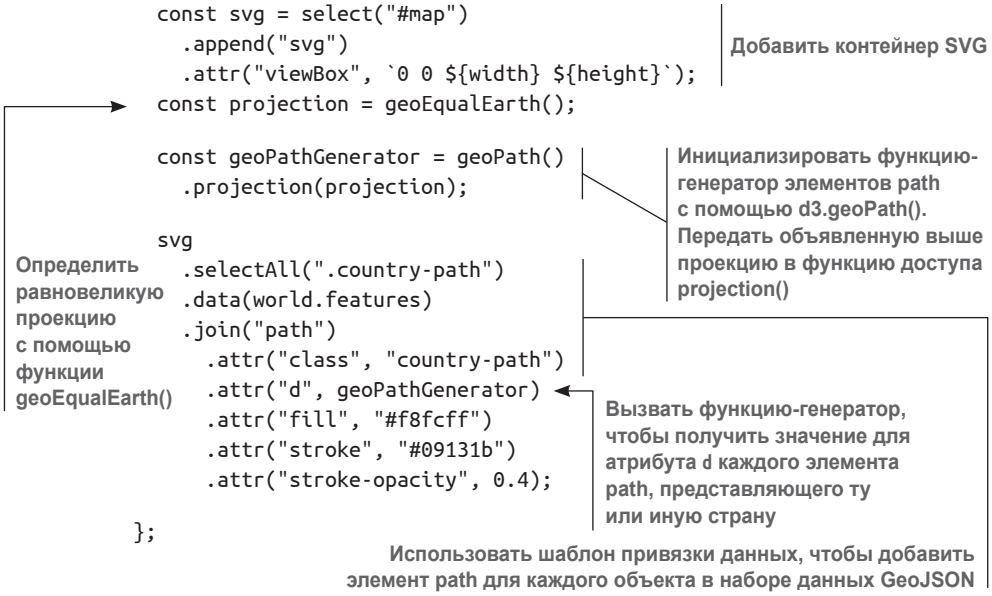
```
import { select } from "d3-selection";
import { geoPath, geoEqualEarth } from "d3-geo";
```

Импортировать необходимые методы D3

```
export const drawWorldMap = (laureates, world) => {

  const width = 1230;
  const height = 620;
```

Объявить размеры карты



Сохранив проект и открыв его в браузере, вы должны увидеть карту мира! Но обратите внимание, что она занимает не все доступное пространство в контейнере SVG и располагается не по центру. Для устранения этих недостатков предназначены функции доступа `transition()` и `scale()`.

Чтобы разместить карту по центру, сместим карту по горизонтали на половину ширины контейнера SVG и по высоте на половину высоты контейнера, как показано ниже. Обратите внимание, что величины смещений передаются в виде массива. В зависимости от используемых геоданных и размера контейнера SVG величины смещений, необходимых для правильного позиционирования карты, могут отличаться, для чего может понадобится подобрать их экспериментальным путем:

```
const projection = geoEqualEarth()
  .translate([width/2, height/2])
```

В отношении масштабирования тоже нет строгих правил, и вам придется попробовать применить разные коэффициенты масштабирования, пока не найдете правильный. В следующем фрагменте мы используем коэффициент масштабирования 220. С этим коэффициентом карта мира должна выглядеть, как показано на рис. 13.5:

```
const projection = geoEqualEarth()
  .translate([width/2, height/2])
  .scale(220);
```



Рис. 13.5 Карта мира в равновеликой проекции, нарисованная на основе данных GeoJSON

Как видите, нарисовать карту на основе данных GeoJSON с помощью D3 довольно просто! Необходимые для этого шаги перечислены на рис. 13.6.

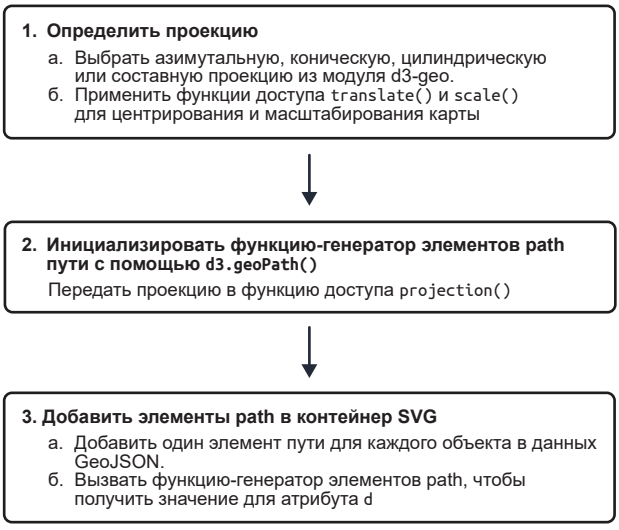


Рис. 13.6 Создание карты на основе данных GeoJSON с помощью D3 выполняется в три шага: определение проекции, инициализация функции генератора элементов path и добавление элементов path в контейнер SVG

13.2.2 Улучшение читаемости с помощью картографической сетки

Картографическая сетка – это сетка линий, представляющих меридианы и параллели на карте. Они помогают лучше понять проекцию

и ее влияние на двумерное представление карты. Они также могут помочь пользователям оценить координаты местоположений.

Неудивительно, что D3 имеет функцию-генератор для создания таких сеток – `d3.geoGraticule()`, – которая возвращает объект GeoJSON. По умолчанию она создает равномерную сетку меридианов и параллелей через каждые 10°, за исключением полюсов, где меридианы отображаются только через каждые 90°. На рис. 13.7 показаны картографические сетки, создаваемые по умолчанию для проекции `d3.geoOrthographic()`, которая напоминает трехмерный глобус.

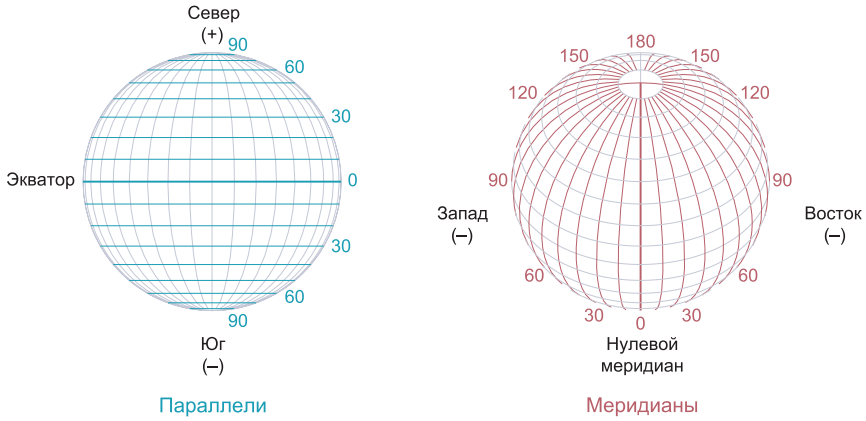


Рис. 13.7 Картографические сетки – это линии, представляющие меридианы и параллели. Линии параллелей параллельны экватору и охватывают 180° от Южного полюса до Северного. Линии меридианов перпендикулярны экватору и охватывают 360° вокруг Земли. Нулевой меридиан, или Гринвичский меридиан, определяется как 0°

Чтобы нарисовать сетку на нашей карте мира, сначала нужно объявить генератор сетки вызовом `d3.geoGraticule()`. Как показано в листинге 13.2, мы сохраняем эту функцию в константе с именем `graticuleGenerator`.

Затем мы добавляем в контейнер SVG элемент группы и сохраняем его в константе `graticules`. Эта группа отвечает за распространение стилей, применяемых к сетке, такие как прозрачность заливки, цвет и ширина обводки.

Вся картографическая сетка рисуется с помощью единственного элемента `path`. В коде мы добавляем этот элемент в `graticules`. Обратите внимание, что для привязки данных к этому элементу вместо `data()` мы используем метод `datum()`. Слово *datum* в английском языке представляет единственное число слова *data* (данные), имеющего множественное число. Этот метод используется для привязки фрагмента данных к одному элементу, и мы передаем ему генератор сетки.

Наконец, мы добавляем еще один элемент `path`, чтобы нарисовать границу вокруг карты. Требуемая система координат доступна с помощью `graticuleGenerator.outline`. В обоих случаях атрибут `d` вычисляется посредством функции-генератора `geoPathGenerator`, инициализированной ранее.

Поскольку нам нужно, чтобы картографическая сетка отображалась под контурами стран, ее элемент `path` следует добавить раньше элементов `path` стран, но после объявления генератора `geoPathGenerator`, потому что он необходим для создания значения атрибута `d`.

Листинг 13.2 Добавление картографической сетки на карту мира (map-world.js)

```
import { select } from "d3-selection";
import { geoPath, geoEqualEarth, geoGraticule } from "d3-geo";

export const drawWorldMap = (laureates, world) => {

  ...

  const projection = geoEqualEarth()

  const geoPathGenerator = geoPath()
    .projection(projection);

  const graticuleGenerator = geoGraticule();

  const graticules = svg
    .append("g")
      .attr("fill", "transparent")
      .attr("stroke", "#09131b")
      .attr("stroke-opacity", 0.2);
  graticules
    .append("path")
    .datum(graticuleGenerator)
      .attr("d", geoPathGenerator);
  graticules
    .append("path")
    .datum(graticuleGenerator.outline)
      .attr("d", geoPathGenerator);

  svg
    .selectAll(".country-path")
    .data(world.features)
    .join("path")
      .attr("class", "country-path")
      .attr("d", geoPathGenerator)
      .attr("fill", "#f8fcff")
      .attr("stroke", "#09131b")
      .attr("stroke-opacity", 0.4);

};
```

Объявить генератор картографической сетки

Добавить элемент группы, который будет содержать элемент path сетки. Использовать его для настройки атрибутов `fill`, `stroke` и `stroke-opacity`

Добавить один элемент path для сетки. Привязать данные к этому элементу с помощью метода datum() и функции graticuleGenerator. Установить атрибут `d` вызовом функции geoPathGenerator, объявленной ранее

Добавить элемент path для отображения границы карты, вызвав метод outline() генератора сетки

После этого шага карта должна выглядеть, как показано на рис. 13.8.



Рис. 13.8 Благодаря добавлению картографической сетки читатель сможет оценить координаты местоположений по широте (горизонтальные линии) и долготе (вертикальные линии) и увидеть, как трехмерный глобус проецируется на двумерную плоскость

13.2.3 Создание картограммы

Картограмма – это карта, которая кодирует данные с использованием разных цветов, обозначающих регионы, такие как страны. Картограммы широко используются для сравнения статистических данных, таких как ВВП стран, численности населения или количества лауреатов Нобелевской премии!

Для создания картограммы нам понадобится цветовая шкала. В нашем проекте количество нобелевских лауреатов, родившихся в разных странах, варьируется от 0 до 289. Страны, где не было рождено ни одного нобелевского лауреата, мы окрасим в белый цвет, а для окрашивания других стран нам понадобятся цвета, охватывающие диапазон от 1 до почти 300. В таких ситуациях очень удобно использовать последовательные цветовые шкалы. Они принимают непрерывную предметную область (количество лауреатов, родившихся в любой стране) и возвращают непрерывный диапазон результатов – значений цвета. В D3 имеются уже готовые последовательные цветовые шкалы в модуле d3-scale-chromatic. В нашем проекте мы используем последовательную многоцветную схему `d3.interpolateYlGnBu`, где `YlGnBu` означает «yellow-green-blue» – «желтый–зеленый–синий».

Эта цветовая шкала показана на рис. 13.9. Страны с небольшим числом лауреатов Нобелевской премии будут отображаться оттенка-

ми желтого, страны с числом лауреатов около 50 будут окрашиваться в бирюзовый цвет, а страны с наибольшим числом – в темно-синий. Мы ограничим верхнюю границу области 100 лауреатами Нобелевской премии. Поскольку только в одной стране число лауреатов значительно больше 100 (США), использование фактического значения 289 в качестве верхнего предела приведет к тому, что почти все остальные страны будут казаться желтыми по сравнению с ними, из-за чего различия между странами станут практически нечитаемыми.



Рис. 13.9 Для представления числа лауреатов Нобелевской премии, родившихся в каждой стране, мы используем последовательную многоцветную шкалу d3.interpolateYlGnBu

В листинге 13.3 показан фрагмент кода из файла `scales.js`, в котором мы инициализируем последовательную цветовую шкалу вызовом метода `d3.scaleSequential()`. Полученная шкала сохраняется в константе `countryColorScale`, экспортируемой для будущего использования. Предметная область шкалы устанавливается как массив с двумя значениями: 1 – наименьшее значение, обозначаемое цветом, и 100 – верхняя граница цветовой шкалы.

Обратите внимание, что диапазон шкалы не задается. Вместо него в `scaleSequential()` передается функция `interpolateYlGnBu`, которая позаботится о возврате нужных цветов.

Листинг 13.3 Объявление цветовой шкалы (scales.js)

```
import { scaleSequential } from "d3-scale";
import { interpolateYlGnBu } from "d3-scale-chromatic";

export const countryColorScale = scaleSequential(interpolateYlGnBu)
  .domain([1, 100]);
```

Чтобы цвета появились на карте, нужно подсчитать количество нобелевских лауреатов, родившихся в каждой стране. В листинге 13.4 показан фрагмент кода из файла `map-world.js`, который перебирает страны в объекте `features` из данных GeoJSON. Как упоминалось выше, объект каждой страны имеет атрибут с именем `properties`, включающий некоторую информацию о стране, например ее название. Это идеальное место для хранения массива нобелевских лауреатов, родившихся в стране. Мы фильтруем массив `laureates` со списком лауреатов и оставляем только тех из них, кто родился в текущей стране.

Листинг 13.4 Добавление лауреатов из массива laureates в каждую страну (map-world.js)

```
export const drawWorldMap = (laureates, world) => {

  world.features.forEach(country => {
    const props = country.properties;
    props.laureates = laureates.filter(laureate =>
      laureate.birth_country === props.name);
  });

  ...

};
```

Далее, импортировав цветовую шкалу, вернемся к коду, где мы добавили элементы `path`, отрисовывающие контуры стран, и вместо прямого присваивания цвета заливки `#f8fcff` атрибуту `fill` используем тернарный оператор для вызова цветовой шкалы, если количество лауреатов больше нуля, как показано в листинге 13.5. Мы передаем количество лауреатов в функцию шкалы, которая возвращает соответствующий цвет.

Листинг 13.5 Применение цвета к каждой стране (map-world.js)

```
import { countryColorScale } from "./scales";

export const drawWorldMap = (laureates, world) => {

  ...

  svg
    .selectAll(".country-path")
    .data(world.features)
    .join("path")
      .attr("class", "country-path")
      .attr("d", geoPathGenerator)
      .attr("fill", d => d.properties.laureates.length > 0
        ? countryColorScale(d.properties.laureates.length)
        : "#f8fcff")
      .attr("stroke", "#09131b")
      .attr("stroke-opacity", 0.4);

};
```

Теперь у нас есть картограмма, изображенная на рис. 13.10. Как видите, наибольшее количество лауреатов Нобелевской премии родилось в США. Далее следуют Великобритания, Германия и Франция. В `index.html` раскомментируйте div с классом `legend-container`. Благодаря этому слева от карты появится цветовая легенда. Мы создали

эту легенду с помощью CSS-свойства `background` и линейного градиента, которые вы найдете в `visualization.css`.

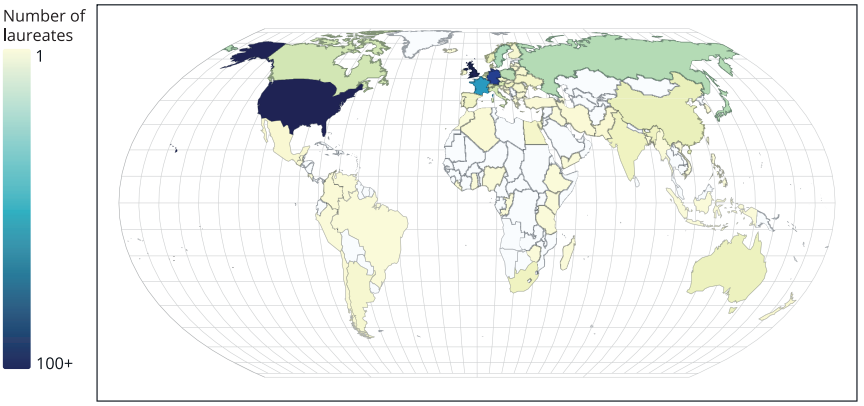


Рис. 13.10 Картограмма, отображающая количество лауреатов Нобелевской премии, родившихся в каждой стране

> **ПРИМЕЧАНИЕ** Помните, что картограммы, несмотря на всю их пользу, подвержены так называемой *проблеме площадных единиц*, которая возникает, когда вы проводите границы или выбираете существующие объекты так, что они непропорционально представляют вашу статистику. Размер географической области может быть намного больше, чем ее значение в данных. Например, Россия имеет гигантскую территорию, по сравнению с Японией. И хотя обе страны имеют примерно одинаковую численность населения, любой цвет, примененный к территории России, привлечет больше внимания, чем цвет, примененный к меньшей территории, такой как территория Японии.

Наша картограмма дает хорошее представление о странах, где родилось больше или меньше лауреатов Нобелевской премии. Но, глядя на нее, мы не можем сказать, сколько именно лауреатов родилось в каждой стране. Поскольку мы работаем с цифровыми визуализациями, то легко можем решить эту проблему, добавив интерактивности! Когда указатель мыши оказывается над страной, мы отобразим подсказку с ее названием и количеством родившихся в ней лауреатов. Файл `index.html` уже содержит `div` с идентификатором `"map-tooltip"`, который позиционирован по абсолютным координатам в правом верхнем углу карты. Именно в него мы вставим текст подсказки.

В листинге 13.6 показаны две функции: `showTooltip()`, которая добавляет нужный текст во всплывающую подсказку и делает ее видимой, и `hideTooltip()`, которая скрывает всплывающую подсказку.

Здесь же мы подключаем обработчики событий к элементам `path` стран. Когда указатель мыши помещается над одним из них, обработчик извлекает нужный текст и вызывает `showTooltip()`. Когда указатель мыши покидает элемент, то вызывается `hideTooltip()`.

Листинг 13.6 Добавление всплывающей подсказки (map-world.js)

```
...
import { transition } from "d3-transition";

export const drawWorldMap = (laureates, world) => {

  ...

  const showTooltip = (text) => {
    select("#map-tooltip")
      .text(text)
      .transition()
      .style("opacity", 1);
    };

  const hideTooltip = () => {
    select("#map-tooltip")
      .transition()
      .style("opacity", 0);
  };

  svg
    .selectAll(".country-path")
    .data(world.features)
    .join("path")
      .attr("class", "country-path")
      .attr("d", geoPathGenerator)
      .attr("fill", d => d.properties.laureates.length > 0
        ? countryColorScale(d.properties.laureates.length)
        : "#f8fcff")
      .attr("stroke", "#09131b")
      .attr("stroke-opacity", 0.4)
      .on("mouseenter", (e, d) => {
        const p = d.properties;
        const lastWord = p.laureates.length > 1
          ? "laureates"
          : "laureate"
        const text = `${p.name}, ${p.laureates.length}
          ${lastWord}`
        showTooltip(text);
      })
      .on("mouseleave", hideTooltip);

};
```

Отображает всплывающую подсказку с соответствующим текстом

Скрывает подсказку

Когда указатель мыши оказывается над элементом path страны, извлечь нужный текст и передать его функции showTooltip()

Когда указатель мыши покидает элемент path страны, вызвать функцию hideTooltip()

На рис. 13.11 показано, как выглядит подсказка, появляющаяся при наведении указателя мыши на Индию.

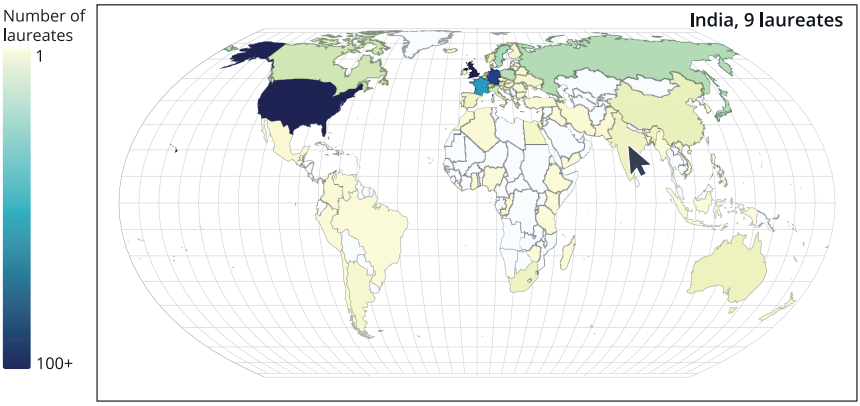


Рис. 13.11 Цветовая шкала помогает определить, в каких странах много нобелевских лауреатов, а в каких мало, но она не позволяет узнать точное их число. Мы делаем эту информацию доступной с помощью простой подсказки

13.2.4 Размещение городов на карте

Проекции используются не только для отображения регионов, но и для размещения отдельных точек на карте. Обычно в своем воображении мы представляем города или людей не как их пространственную форму (хотя иногда мы так поступаем с особенно большими городами), а как точку на карте, размер которой зависит, например, от численности населения. Мы можем использовать проекцию D3 для вычисления местоположения города, передав ей массив с долготой и широтой и получив соответствующие координаты внутри контейнера SVG.

Допустим, нам нужно отобразить на нашей карте города, в которых родились нобелевские лауреаты. Раскомментируйте `div` с идентификатором `"filters-container"` в `index.html`, чтобы увидеть радиокнопки для выбора способа отображения лауреатов – по странам или по городам.

В листинге 13.7 мы обрабатываем событие выбора радиокнопки и вызываем функцию, которая отображает страны или города. Мы также реорганизовали ранее написанный код, чтобы добавить возможность переключения между двумя состояниями карты. Когда страница загружается, мы вызываем функцию `displayCountries()`, отвечающую за применение цветовой шкалы к странам.

Листинг 13.7 Рефакторинг с целью реализации динамического выбора стиля картограммы (map-world.js)

```
export const drawWorldMap = (laureates, world) => {

  ...

  let isCountryMap = true;

  svg
    .selectAll(".country-path")
    .data(world.features)
    .join("path")
      .attr("class", "country-path")
      .attr("d", geoPathGenerator)
      .attr("stroke", "#09131b")
      .attr("stroke-opacity", 0.4);

  const updateCountryFills = () => {
    selectAll(".country-path")
      .on("mouseenter", (e, d) => {
        const p = d.properties;
        const lastWord = p.laureates.length > 1 ? "laureates" : "laureate";
        const text = `${p.name}, ${p.laureates.length} ${lastWord}`;
        showTooltip(text);
      })
      .on("mouseleave", hideTooltip)
      .transition()
      .attr("fill", d => d.properties.laureates.length > 0
        ? countryColorScale(d.properties.laureates.length)
        : "#f8fcff");
  };

  const displayCountries = () => {
    isCountryMap = true;
    updateCountryFills();
  };

  const displayCities = () => {};

  selectAll("input#countries, input#cities")
    .on("click", e => {
      if (e.target.id === "countries") {
        displayCountries();
      } else if (e.target.id === "cities") {
        displayCities();
      }
    });

  displayCountries();
};
```

Инициализировать переменную состояния, которая определяет, что на карте – страны или города, где родились лауреаты

По событию выбора той или иной радиокнопки вызвать соответствующую функцию

При загрузке страницы вызвать функцию, отображающую страны с лауреатами

Название города, где родился каждый нобелевский лауреат, доступно в массиве `laureates` в атрибуте `birth_city`. А в атрибутах `birth_city_latitude` и `birth_city_longitude` содержатся широта и долгота города соответственно. Прежде чем отобразить эти города на карте, мы создадим новый массив `cities`, в который поместим все города, где родились нобелевские лауреаты, их широты, долготы и имена лауреатов, родившихся в них, в виде массива. В листинге 13.8 показано, как это сделать.

Листинг 13.8 Создание массива cities (map-world.js)

```
export const drawWorldMap = (laureates, world) => {

  const cities = [];
  laureates.forEach(laureate => {
    if (laureate.birth_country !== "" &&
        laureate.birth_city !== "") {

      const relatedCity = cities.find(city =>
        city.city === laureate.birth_city &&
        city.country === laureate.birth_country);

      if (relatedCity) {
        relatedCity.laureates.push(laureate);
      } else {
        cities.push({
          city: laureate.birth_city,
          country: laureate.birth_country,
          latitude: laureate.birth_city_latitude,
          longitude: laureate.birth_city_longitude,
          laureates: [laureate]
        });
      }

    }
  });

  ...

};
```

Создать пустой массив с именем cities

Выполнить обход элементов массива laureates и выбрать из него города и страны рождения лауреатов. Некоторые лауреаты – это международные организации, которые не привязаны к определенному месту рождения, поэтому информация о них не попадает в массив cities

Если массив cities уже содержит объект, соответствующий городу рождения текущего лауреата, то добавить нового лауреата в массив laureates. Иначе создать новый объект city и поместить его в массив cities

Теперь мы готовы показать города на карте! Добавим круг в контейнер SVG для каждого города в массиве `cities` и зададим его радиус пропорционально количеству лауреатов, родившихся в этом городе. В листинге 13.9 показан фрагмент кода из файла `scales.js`, где мы объявили функцию `getCityRadius()`, получающую два числа: количество лауреатов и максимально возможное количество лау-

реатов. Второе число используется для инициализации радиальной шкалы. В ответ функция возвращает радиус, соответствующий количеству лауреатов.

Листинг 13.9 Объявление радиальной шкалы (scales.js)

```
import { scaleSequential, scaleRadial } from "d3-scale";
import { interpolateYlGnBu } from "d3-scale-chromatic";

export const countryColorScale = scaleSequential(interpolateYlGnBu)
  .domain([1, 100]);

export const getCityRadius = (numLaureates, maxLaureates) => {
  const cityRadiusScale = scaleRadial()
    .domain([0, maxLaureates])
    .range([0, 25]);

  return cityRadiusScale(numLaureates);
};
```

Для вычисления местоположения каждого города на карте можно использовать проекцию, созданную в разделе 13.2.1, как показано ниже. Допустим, у нас есть функция проекции, хранящаяся в константе с именем `projection`. Если передать в функцию проекции массив с долготой и широтой, то в ответ она вернет массив соответствующих координат внутри контейнера SVG:

```
const projection = d3.geoEqualEarth();

[x, y] = projection([longitude, latitude]);
```

Ниже приводится довольно длинный листинг 13.10, но код в нем относительно прост, если разбить его на более мелкие части. Он содержит четыре основные функции:

- `updateCountryFills()` добавляет обработчики событий в режиме отображения численности лауреатов по странам и установку цвета их заливки;
- `updateCityCircles()` добавляет обработчики событий в режим отображения численности лауреатов по городам и установку радиусов кругов, соответствующих городам;
- `displayCountries()` вызывается при загрузке страницы и выборе радиокнопки переключения в режим отображения по странам. Она удаляет круги городов и вызывает `updateCountryFills()`;
- `displayCities()` вызывается при выборе радиокнопки переключения в режим отображения по городам. Удаляет обработ-

чики событий из элементов `path` стран и присваивает их атрибуту `fill` белый цвет. Затем добавляет круги городов и задает их позиции, передавая долготу и широту в проекцию, после чего вызывает `updateCityCircles()`.

Первые две функции могут показаться ненужными, но они понадобятся нам позже, когда мы добавим функционал выделения мышью.

Листинг 13.10 Отображение стран или городов (map-world.js)

```
...
import { select, selectAll } from "d3-selection";
import { max } from "d3-array";
import { countryColorScale, getCityRadius } from "./scales";

export const drawWorldMap = (laureates, world) => {
  ...

  const updateCountryFills = () => {
    selectAll(".country-path")
      .on("mouseenter", (e, d) => {
        const p = d.properties;
        const lastWord = p.laureates.length > 1 ? "laureates" : "laureate";
        const text = `${p.name}, ${p.laureates.length} ${lastWord}`;
        showTooltip(text);
      })
      .on("mouseleave", hideTooltip)
      .transition()
      .attr("fill", d => d.properties.laureates.length > 0
        ? countryColorScale(d.properties.laureates.length)
        : "#f8fcff");
  };
  const maxLaureatesPerCity = max(cities, d => d.laureates.length);
  const updateCityCircles = () => {
    selectAll(".circle-city")
      .on("mouseenter", (e, d) => {
        const lastWord = d.laureates.length > 1 ? "laureates" : "laureate";
        const text = `${d.city}, ${d.laureates.length} ${lastWord}`;
        showTooltip(text);
      })
      .on("mouseleave", hideTooltip)
      .transition()
      .attr("r", d => getCityRadius(d.laureates.length,
        maxLaureatesPerCity));
  };
```

Добавить обработчики событий в режиме отображения по странам и задать цвета их заливки

Добавить круги городов и их обработчики событий

```
  const displayCountries = () => {
    isCountryMap = true;

    selectAll(".circle-city")
      .transition()
      .attr("fill-opacity", 0)
      .attr("stroke-opacity", 0)
      .remove();

    updateCountryFills();
  };
```

Когда выбрано отображение по странам, скрыть круги городов и вызвать функцию, отвечающую за установку цветов заливки стран

```
  const displayCities = () => {
    isCountryMap = false;

    selectAll(".country-path")
      .on("mouseenter", null)
      .on("leave", null)
      .transition()
      .attr("fill", "#f8fcff");

    selectAll(".circle-city")
      .data(cities)
      .join("circle")
        .attr("class", "circle-city")
        .attr("cx", d => projection([d.longitude,
          d.latitude])[0])
        .attr("cy", d => projection([d.longitude,
          d.latitude])[1])
        .attr("fill", "#35a7c2")
        .attr("fill-opacity", 0.5)
        .attr("stroke", "#35a7c2");

    updateCityCircles();
};
```

Когда выбрано отображение по городам, удалить обработчики событий из элементов path стран и задать белый цвет их заливки

Использовать шаблон привязки данных, чтобы добавить круг для каждого города

Получить местоположение каждого круга, вызвав функцию проекции и передав ей долготу и широту города в массиве

```
  selectAll("input#countries, input#cities")
    .on("click", e => {
      if (e.target.id === "countries") {
        displayCountries();
      } else if (e.target.id === "cities") {
        displayCities();
      }
    });

  displayCountries();

};
```

Завершая визуализацию, добавим отображение цветовой легенды, когда выбран режим отображения по странам, и легенды с радиусами, когда выбран режим отображения по городам. Мы уже создали легенду с радиусами. Нам остается только вызвать функцию `drawLegend()` и передать ей максимальное количество лауреатов для каждого города, как показано в листинге 13.11.

Листинг 13.11 Динамическое отображение легенд (map-world.js)

```
...
import { drawLegend } from "./legend";

export const drawWorldMap = (laureates, world) => {

  ...

  const displayCountries = () => {

    ...

    select(".legend-cities")
      .style("display", "none");
    select(".legend-countries")
      .style("display", "flex");
  };

  const displayCities = () => {

    ...

    select(".legend-countries")
      .style("display", "none");
    select(".legend-cities")
      .style("display", "block");
  };

drawLegend(maxLaureatesPerCity);

};
```

Мы реализовали отображение городов на карте, используя их долготу и широту! Теперь вы можете переключаться между двумя режимами отображения с помощью радиокнопок. Как нетрудно заметить на рис. 13.12, круги городов расположены очень плотно, особенно в Европе. Функция выделения мышью, которую мы добавим позже, поможет сделать карту городов более читаемой.

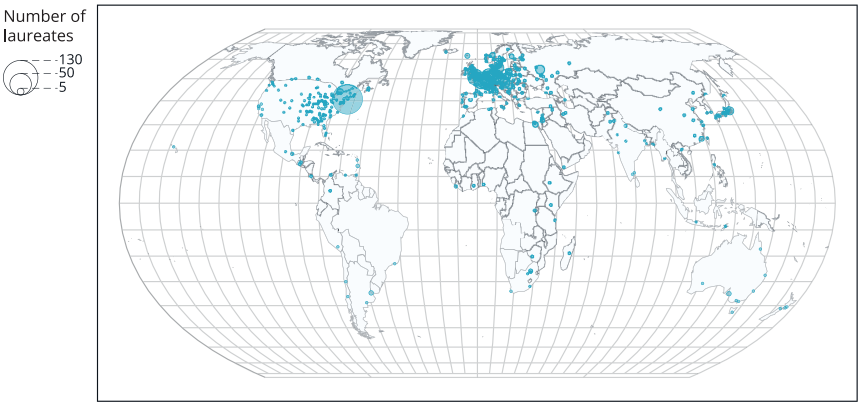


Рис. 13.12 Города, где родились нобелевские лауреаты

13.3 Масштабирование и панорамирование

Карты широко распространены в интернете, и пользователи ожидают от них определенного уровня интерактивности, например возможности масштабирования и панорамирования. Масштабирование и панорамирование позволяют пользователям сосредоточиться на интересующей их области и увидеть больше деталей. В этом разделе мы реализуем функцию масштабирования и дадим пользователю возможность изменять масштаб карты и перемещать ее.

В D3 имеется модуль d3-zoom, содержащий функции масштабирования и панорамирования, однако реализация этих функций не так проста, как кажется. В листинге 13.12 мы объявляем функцию масштабирования, которая использует метод `d3.zoom()`, и сохраняем ее в константе с именем `zoomHandler`. Мы также добавляем в zoomHandler обработчик события `"zoom"`. Когда появляется это событие, мы выводим информацию о нем в консоль.

Для поддержки масштабирования мы выбираем `div` с классом `"map-container"` (первый родитель контейнера SVG) и вызываем метод `call()`, чтобы подключить `zoomHandler` к выборке.

Листинг 13.12 Реализация масштабирования и панорамирования (map-world.js)

```
...
import { zoom } from "d3-zoom";

export const drawWorldMap = (laureates, world) => {

  ...

  const zoomHandler = zoom()
```

Создать новую функцию, реализующую масштабирование

```
    .on("zoom", (e) => {
      console.log(e);
    });
  select(".map-container")
    .call(zoomHandler);

};
```

Добавить обработчик события масштабирования, который выводит информацию о событии в консоль

Применить поведение масштабирования к выборке контейнера карты с помощью метода call()

Теперь сохраните проект, откройте его в браузере, поместите указатель мыши над картой и начните масштабировать карту, вращая колесико мыши или разводя пальцы на трекпаде. Вы не увидите никакой реакции, как, впрочем, и ожидалось, потому что мы еще не применили никаких преобразований к карте. Но, открыв консоль в браузере, вы должны увидеть, что событие масштабирования фиксируется нашей картограммой, как показано на рис. 13.13. Это событие содержит объект `transform` с коэффициентом масштабирования `k`, перемещением по горизонтали `x` и перемещением по вертикали `y`.

```
▼ ZoomEvent {type: 'zoom', sourceEvent: WheelEvent, transform: Transform, _: Dispatch, target: f}
  ▶ sourceEvent: WheelEvent {isTrusted: true, deltaX: -0, deltaY: -2.7549855709075928, deltaZ: 0, deltaMode: 0, …}
  ▶ target: f zoom(selection)
  ▶ transform: Transform {k: 1.8420359696923372, x: 272610.815749858, y: -308257.07603286544}
    type: "zoom"
```

Рис. 13.13 Событие масштабирования содержит объект transform с атрибутами k (коэффициент масштабирования), x (перемещение по горизонтали) и y (перемещение по вертикали)

Настроив получение событий, мы можем использовать объект `transform` для масштабирования и перемещения карты. В листинге 13.13 мы обновляем атрибут `transform` контейнера SVG на основе этого события.

Листинг 13.13 Масштабирование карты (map-world.js)

```
export const drawWorldMap = (laureates, world) => {

  ...

  const zoomHandler = zoom()
    .on("zoom", (e) => {
      console.log(e);
      svg.attr("transform", e.transform);
    });

  select(".map-container")
    .call(zoomHandler);

};
```

Использовать полученное событие масштабирования для обновления атрибута `transform` контейнера SVG

Теперь масштабирование и панорамирование карты в браузере должно работать. Но обратите внимание, что карта выходит за пределы области, ограниченной черной границей, что выглядит неаккуратно. К счастью, это легко исправить! Откройте `visualization.css` и найдите стили, применяемые к выборке `.map-container` – первому родителю контейнера SVG. Добавьте свойство `overflow` со значением `hidden`, и теперь карта не будет отображаться за границами:

```
.map-container {
  position: relative;
  border: 1px solid #09131b;
  overflow: hidden;
}
```

Следующим шагом мы должны ограничить масштабирование и перемещение карты пользователем, чтобы избежать ситуации, когда карта полностью окажется вне поля зрения. Поддержка масштабирования имеет две удобные функции доступа, `scaleExtent()` и `translateExtent()`, которые, судя по их именам, позволяют устанавливать пределы масштабирования и перемещения соответственно. Эти функции принимают массив с минимальным и максимальным значениями. В листинге 13.14 мы устанавливаем минимальный масштаб равным 1 (т. е. 100 %), а максимальный равным 5 (500 %). Мы также устанавливаем максимальное перемещение равным половине ширины карты по горизонтали и половине ее высоты по вертикали.

Листинг 13.14 Ограничение пределов масштабирования и перемещения (map-world.js)

```
export const drawWorldMap = (laureates, world) => {

  ...

  const zoomHandler = zoom()
    .scaleExtent([1, 5])
    .translateExtent([[-width/2, -height/2],
      [3*width/2, 3*height/2]])
    .on("zoom", (e) => {
      console.log(e);
      svg.attr("transform", e.transform);
    });

  select(".map-container")
    .call(zoomHandler);

};
```

Разрешите пользователю масштабировать карту в 1–5 раз от ее исходного размера

Ограничить перемещение карты так, чтобы половина ее ширины и высоты всегда находилась в пределах установленных границ

Также было бы неплохо предусмотреть возможность быстро вернуть карту в исходное состояние. Для этого в `index.html` раском-

ментируйте кнопку с идентификатором `"map-reset"`. После этого в правом верхнем углу появится синяя кнопка с текстом **Reset Zoom** (Сбросить масштаб).

Поскольку нет смысла показывать эту кнопку до выполнения масштабирования, мы будем добавлять и удалять имя класса `"hidden"` в кнопку `reset`, когда это уместно. Объекты с классом `"hidden"` получают свойство `opacity` со значением 0 и не принимают события щелчков на них (см. `visualization.css`). Сразу после загрузки страницы мы добавляем класс `"hidden"` в кнопку `map-reset`, а затем, когда перехватывается событие масштабирования, проверяем, имеет ли эта кнопка класс `"hidden"`, и если имеет, то удаляем его.

Нажатие кнопки должно вызывать фактический сброс масштаба. Как показано в листинге 13.15, для этого мы добавляем обработчик события `click` в кнопку сброса, который вызывает функцию доступа `transform()` объекта `zoomHandler`, присоединенного к выборке `.map-container`. Во втором параметре мы передаем нужное нам значение преобразования, в данном случае результат вызова функции `d3.zoomIdentity`. Эта функция описывает преобразование с масштабом, равным 1 (т. е. 100 %), и смещением 0.

Листинг 13.15 Обработка кнопки сброса (map-world.js)

```
...
import { zoom, zoomIdentity } from "d3-zoom";

export const drawWorldMap = (laureates, world) => {

  ...

  const zoomHandler = zoom()
    .scaleExtent([1, 5])
    .translateExtent([[-width/2, -height/2], [3*width/2, 3*height/2]])
    .on("zoom", (e) => {
      console.log(e);
      svg.attr("transform", e.transform);

      if (select("#map-reset").classed("hidden")) {
        select("#map-reset")
          .classed("hidden", false);
      }
      if (e.transform.k === 1 &&
          e.transform.x === 0 &&
          e.transform.y === 0) {
        select("#map-reset")
          .classed("hidden", true);
        }
    });

  select(".map-container")
```

Если кнопка сброса имеет имя класса "hidden", то удалите этот класс, чтобы сделать кнопку видимой

Если карта возвращается в исходное состояние, то нужно скрыть кнопку сброса

```
    .call(zoomHandler);

  select("#map-reset")
    .attr("class", "hidden")
    .on("click", () => {
      select(".map-container")
        .transition()
        .call(zoomHandler.transform, zoomIdentity);
    });

};
```

Скрыть кнопку сброса

При нажатии кнопки сброса вернуть масштаб в исходное состояние

Если после масштабирования нажать кнопку **Reset Zoom** (Сбросить масштаб), то карта должна вернуться в исходное состояние.

Семантическое масштабирование

Говоря об увеличении масштаба объекта, вы, естественно, думаете об увеличении его размеров. Однако при работе с картограммами можно не только увеличить размер или разрешение, но также изменять тип отображаемых данных. Такая функциональность известна как *семантическое масштабирование*, в отличие от графического масштабирования, реализованного в этом разделе.

Например, когда вы смотрите на уменьшенную карту, то видите лишь границы стран и несколько крупных городов, а при увеличении масштаба вы видите дороги, небольшие города, парки и т. д. В реализациях карт на мобильных устройствах часто используется такое семантическое масштабирование. Однако семантическое масштабирование можно реализовать в любых видах визуализаций, а не только карт. Оно позволяет представлять стратегическую или глобальную информацию при маленьком масштабе и более детальные сведения при увеличении масштаба.

13.4 Добавление функции выделения мышью

Другой вид интерактивного поведения, который хорошо сочетается с картами, – это выделение. *Выделение* – это одно- или двухмерная область, созданная нажатием кнопки мыши и перетаскиванием ее указателя. Функция выделения мышью часто используется для выбора элементов, масштабирования области или фильтрации данных по временной шкале.

Наш текущий проект визуализирует данные, накопленные в период с 1901 по 2022 год. Но что, если мы хотим узнать, изменились ли пропорции рождения лауреатов Нобелевской по странам по сравнению с началом XX века? Обрабатывая события выделения мышью,

можно позволить пользователям выбирать определенный диапазон лет на временной шкале. Для этого воспользуемся методами из модуля d3-brush. В листинге 13.16 мы просто используем ось D3 для создания временной шкалы, которая будет служить основой для нашей функции выделения.

Листинг 13.16 Добавление временной шкалы (map-world.js)

```
...
import { max, min } from "d3-array";
import { scaleLinear } from "d3-scale";
import { axisBottom } from "d3-axis";
import { format } from "d3-format";

export const drawWorldMap = (laureates, world) => {

  ...

  const tlWidth = 1000;
  const tlHeight = 80;
  const tlMargin= { top: 0, right: 10, bottom: 0, left: 0 };
  const tlInnerWidth = tlWidth - tlMargin.right -
    tlMargin.left;

  const minYear = min(laureates, d => d.year);
  const maxYear = max(laureates, d => d.year);
  const xScale = scaleLinear()
    .domain([minYear, maxYear])
    .range([0, tlInnerWidth]);
  const yearsSelector = select("#years-selector")
    .append("svg")
      .attr("viewBox", `0 0 ${tlWidth} ${tlHeight}`);

  const xAxisGenerator = axisBottom(xScale)
    .tickFormat(format(""))
    .tickSizeOuter(0);
  yearsSelector
    .append("g")
      .attr("class", "axis-x")
      .attr("transform", `translate(0, 30)`)
      .call(xAxisGenerator);

};
```

Установить размеры временной шкалы

Объявить линейную шкалу, отвечающую за распределение годов по доступному горизонтальному пространству

Добавить контейнер SVG в div с идентификатором "years-selector"

Объявить генератор нижней оси

Добавить ось в контейнер SVG

Только что созданная временная шкала должна выглядеть, как показано на рис. 13.14.

Как показано в листинге 13.17, функциональность выделения мышью в D3 реализуется подобно масштабированию. Сначала мы создаем одномерное выделение вызовом `d3.brushX()` и сохраняем его в константе с именем `brushHandler`. С помощью функции доступа

`extent()` сообщаем библиотеке D3, какая область доступна для этого выделения: от верхнего левого до нижнего правого угла временной шкалы. Затем подключаем обработчик события `"brush"`, который вызывает функцию `handleBrush()`. На данный момент `handleBrush()` просто выводит информацию о событии в консоль, но позднее, поближе познакомившись с этим событием, мы добавим все необходимые действия. Кроме того, нам нужно прикрепить функцию `brushHandler` к выборке временной шкалы, и мы инициализируем выделение так, чтобы по умолчанию оно охватывало всю временную шкалу.

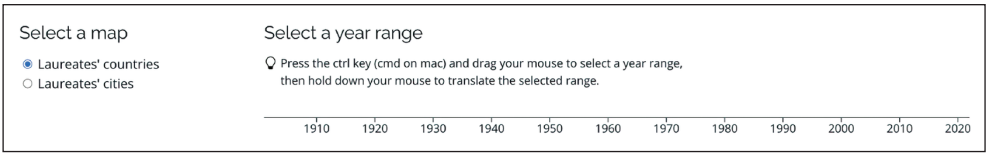


Рис. 13.14 Временная шкала, которая будет играть роль селектора диапазона годов

Листинг 13.17 Добавление функциональности выделения указателем мыши (map-world.js)

```
...
import { brushX } from "d3-brush";

export const drawWorldMap = (laureates, world) => {

  ...

  const handleBrush = (e) => {
    console.log(e);
  };
  const brushHandler = brushX()
    .extent([[0, 0], [tlInnerWidth, tlHeight]])
    .on("brush", handleBrush);

  yearsSelector
    .call(brushHandler)
    .call(brushHandler.move, [xScale(minYear),
      xScale(maxYear)]);

};
```

Установить пределы, в которых может обнаруживаться событие выделения, от верхнего левого угла до нижнего правого угла временной шкалы

Создать новое одномерное выделение по оси x

Эта функция вызывается при появлении события выделения мышью

При появлении события выделения вызвать функцию brushHandler

Добавить обработку выделения в выборку с временной шкалой

Инициализировать область выделения по умолчанию, охватывающую всю временную шкалу

После добавления этого кода временная шкала должна оказаться внутри закрашенной области синего цвета с серыми прямоугольниками вертикальных маркеров по бокам (см. рис. 13.15).

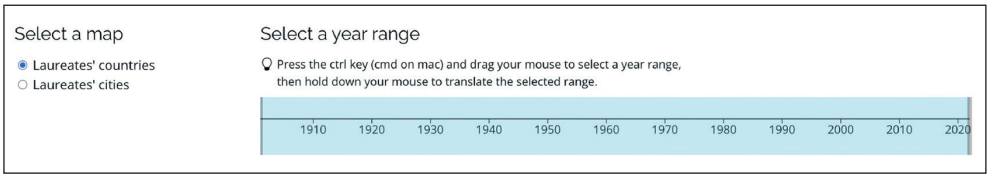


Рис. 13.15 Синяя область выделения с серыми прямоугольниками вертикальных маркеров по бокам по умолчанию охватывает весь диапазон годов

Эта синяя область представляет выбранный диапазон годов. Если вы откроете инструмент инспектора в браузере и посмотрите на разметку, то увидите, что D3 добавила четыре прямоугольника поверх временной шкалы:

- `rect.overlay` покрывает всю область, которую можно выделить;
- `rect.selection` охватывает текущее выделение. Мы установили выделение, по умолчанию охватывающее весь диапазон, но его можно изменить, перетаскивая указателем мыши маркеры слева и справа;
- `rect.handle--w` – это маркер с левой стороны выделения;
- `rect.handle--e` – это маркер с правой стороны выделения.

Файл `visualization.css` содержит следующие стили, делающие последние три прямоугольника узнаваемыми:

```
.selection {
  fill: #35a7c2;
  fill-opacity: 0.3;
  stroke: none;
}
.handle {
  fill: #09131b;
  fill-opacity: 0.3;
}
```

Теперь сгенерируйте событие выделения, ухватив мышью маркер слева или справа и перетащив его вдоль по временной шкале. Также есть возможность перетащить мышью влево или вправо всю область выделения целиком. Посмотрите в консоли, какую информацию несет событие `brush`. Как показано на рис. 13.16, оно содержит массив `selection` с горизонтальными координатами левой и правой границ выделения. Если бы мы использовали `d3.brush()` вместо `d3.brushX()`, то получили бы координаты `x` и `y` верхнего левого и нижнего правого углов выделенной области.

Мы можем использовать координаты в массиве `selection` для определения соответствующих годов. Непрерывные шкалы, такие как линейная шкала, которую мы использовали для рисования временной шкалы, имеют метод `invert()`, принимающий значение из

выходного диапазона и возвращающий соответствующее значение из предметной области, как показано в листинге 13.18. Этот метод применяется для выполнения обратного преобразования. Здесь мы округляем минимальное и максимальное значения области выбора, потому что нам не нужно, чтобы годы были дробными числами, и обновляем две переменные с именами `brushMin` и `brushMax`. Мы используем их чуть ниже, чтобы на основе сделанного выбора обновить цвета стран или круги городов. Но в начальный момент эти переменные получают минимальное и максимальное значения, соответствующие начальному и конечному годам на временной шкале, т. е. области выбора по умолчанию.

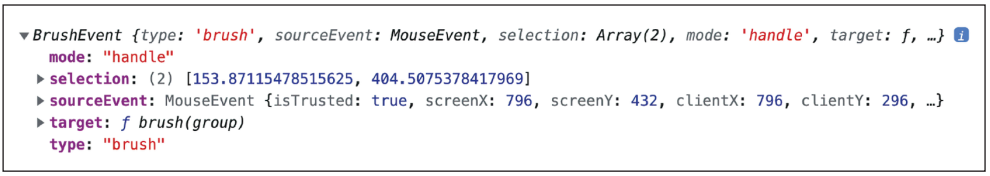


Рис. 13.16 Событие brush несет массив selection с горизонтальными координатами левой и правой границ выделения

Листинг 13.18 Добавление функциональности выделения мышью (map-world.js)

```
...
import { brushX } from "d3-brush";

export const drawWorldMap = (laureates, world) => {

  ...

const minYear = min(laureates, d => d.year);
const maxYear = max(laureates, d => d.year);
let brushMin = minYear;
let brushMax = maxYear;

  const handleBrush = (e) => {
    console.log(e);

    brushMin = Math.round(xScale.invert(e.selection[0]));
    brushMax = Math.round(xScale.invert(e.selection[1]));
    if (isCountryMap) {
      updateCountryFills();
    } else {
      updateCityCircles();
    }
  };

  const brushHandler = brushX()
```

Использовать функцию invert() шкалы xScale для определения начального и конечного годов в выбранной области

Если в данный момент выбрана карта отображения по странам, то обновить цвета страны. Иначе обновить круги городов

```
      .extent([[0, 0], [tlInnerWidth, tlHeight]])
      .on("brush", handleBrush);

    yearsSelector
      .call(brushHandler)
      .call(brushHandler.move, [xScale(minYear), xScale(maxYear)]);

  };
```

В листинге 13.19 мы вызываем функцию `updateCountryFills()` или `updateCityCircles()`, чтобы на основе выбора обновить цвет стран или размеры кругов, представляющих города. При обновлении цветов стран сначала создается полная копия набора данных GeoJSON, чтобы избежать изменения оригинала. Затем отфильтровываются лауреаты, не попавшие в выбранные годы, после чего обновляются данные, привязанные к элементам `path` стран. Аналогично действует функция, обновляющая круги городов, создавая полную копию и фильтруя массив `cities`.

Листинг 13.19 Обновление цветов стран и кругов городов (map-world.js)

```
...
import { brushX } from "d3-brush";

export const drawWorldMap = (laureates, world) => {

  ...

  const updateCountryFills = () => {
    const selectedData = JSON.parse(
      JSON.stringify(world.features));
    selectedData.forEach(d => {
      if (d.properties.laureates) {
        d.properties.laureates = d.properties
          .laureates.filter(l => l.year >= brushMin &&
           l.year <= brushMax);
      }
    });

    selectAll(".country-path")
      .data(selectedData)
      .on("mouseenter", (e, d) => {
        const p = d.properties;
        const lastWord = p.laureates.length > 1 ? "laureates" : "laureate";
        const text = `${p.name}, ${p.laureates.length} ${lastWord}`;
        showTooltip(text);
      })
      .on("mouseleave", hideTooltip)
      .transition()
      .attr("fill", d => d.properties.laureates.length > 0
        ? countryColorScale(d.properties.laureates.length)
```

Создать полную копию данных GeoJSON, чтобы избежать изменения оригинала

Отфильтровать лауреатов, не попавших в выбранные годы

Обновить данные, привязанные к элементам path стран

```
        : "#f8fcff");
  };

  const maxLaureatesPerCity = max(cities, d => d.laureates.length);
  const updateCityCircles = () => {
    const selectedData = JSON.parse(
      JSON.stringify(cities));
    selectedData.forEach(city => {
      city.laureates = city.laureates.filter(l =>
        l.year >= brushMin && l.year <= brushMax);
    });

    selectAll(".circle-city")
      .data(selectedData)
      .on("mouseenter", (e, d) => {
        const lastWord = d.laureates.length > 1 ? "laureates" : "laureate";
        const text = `${d.city}, ${d.laureates.length} ${lastWord}`;
        showTooltip(text);
      })
      .on("mouseleave", hideTooltip)
      .transition()
      .attr("r", d => getCityRadius(d.laureates.length,
        maxLaureatesPerCity));
  };

};
```

Создать полную копию массива cities, чтобы избежать изменения оригинала

Отфильтровать лауреатов, не включенных в выбранные годы

Обновить данные, привязанные к кругам городов

Функция выбора мышью готова! Проверьте ее с вариантами отображения по странам и городам.

13.5 Рисование карты на основе данных TopoJSON

Мы уже знаем, что географические данные распространяются в двух основных форматах – GeoJSON и TopoJSON. До сих пор мы работали только с форматом GeoJSON, но в этом разделе мы используем данные в формате TopoJSON и нарисуем карту 96 департаментов Франции.

Чтобы нарисовать карту на основе данных TopoJSON, нужно установить клиентскую библиотеку TopoJSON (www.npmjs.com/package/topojson-client), которая преобразует данные в формат, который библиотека D3 сможет прочитать и обработать. Чтобы установить библиотеку в проект, откройте окно системного терминала и выполните команду `npm i topojson-client`. Затем откройте файл `map-france.js` и найдите функцию `drawFranceMap()`, с которой мы продолжим работу далее.

В листинге 13.20 показан код, преобразующий данные из формата TopoJSON в формат GeoJSON вызовом метода `feature()` из библио-

теки `topojson-client`. Этот метод принимает два аргумента: набор данных `topojson`, также называемый *топологией*, и объект с набором геометрических фигур, который доступен как атрибут `object.FRA_adm2`. Полученный результат в формате GeoJSON сохраняется в константе с именем `departments`.

Нам также нужно извлечь границы отдельно, что мы и делаем вызовом метода `mesh()`. Этот метод возвращает GeoJSON-объект `MultiLineString`. Мы используем его чуть ниже для рисования границ.

Листинг 13.20 Извлечение данных из набора в формате TopoJSON (map-france.js)

```
import * as topojson from "topojson-client";

export const drawFranceMap = (laureates, france) => {

  let departments = topojson.feature(france,
    france.objects.FRA_adm2).features;

  let borders = topojson.mesh(france,
    france.objects.FRA_adm2);

};
```

Преобразовать данные TopoJSON в объекты GeoJSON

Извлечь границы департаментов

После этого рисование карты мало чем отличается от рисования на основе данных GeoJSON. В листинге 13.21 мы объявляем новую проекцию Меркатора с помощью `d3.geoMercator()`, затем создаем генератор элементов `path`. После этого используем объект `departments`, сгенерированный на предыдущем шаге, чтобы добавить элементы `path` департаментов в контейнер SVG. Мы не присваиваем этим элементам атрибут `stroke`, а добавляем дополнительный элемент `path` и устанавливаем его атрибут `d`, передавая объект `borders` генератору `geoPathGenerator`, чтобы создать границы. Таким образом, границы будут нарисованы с помощью одного элемента `path` без перекрытий.

Листинг 13.21 Рисование карты (map-france.js)

```
import * as topojson from "topojson-client";
import { select } from "d3-selection";
import { geoMercator, geoPath } from "d3-geo";

export const drawFranceMap = (laureates, france) => {

  let departments = topojson.feature(france,
    france.objects.FRA_adm2).features;

  let borders = topojson.mesh(france, france.objects.FRA_adm2);

  const width = 800;
  const height = 800;
```

Инициализировать размеры карты

```
const projection = geoMercator()
  .scale(3000)
  .translate([280, 3150]);
```

Создать проекцию Меркатора

```
const geoPathGenerator = geoPath()
  .projection(projection);
```

Инициализировать генератор элементов path

```
const svg = select("#map-france")
  .append("svg")
    .attr("viewBox", `0 0 ${width} ${height}`);
```

Добавить контейнер SVG в div с идентификатором "map-france"

```
svg
  .selectAll(".department")
  .data(departments)
  .join("path")
    .attr("class", "department")
    .attr("d", d => geoPathGenerator(d))
    .attr("fill", "#f8fcff");
```

Использовать шаблон привязки данных, чтобы добавить элемент path для каждого департамента. Вызвать geoPathGenerator(), чтобы задать атрибут d

```
svg
  .append("path")
    .attr("class", "departments-borders")
    .attr("d", geoPathGenerator(borders))
    .attr("fill", "none")
    .attr("stroke", "#09131b")
    .attr("stroke-opacity", 0.4);
```

Добавить один элемент path для границ и передать объект borders в geoPathGenerator(), чтобы задать его атрибут d

```
};
```

В результате должна появиться карта Франции, как показано на рис. 13.17.

Как объяснялось в начале главы, одно из преимуществ TopoJSON – отсутствие избыточности в данных. Каждая общая граница представлена одной дугой, а не двумя отдельными кривыми. Такой подход не только уменьшает размер файлов, но и делает карты намного чище: общие границы рисуются только один раз. На рис. 13.18 показано, как выглядит результат двойного рисования общих границ на карте мира GeoJSON по сравнению с аккуратными границами на карте Франции.

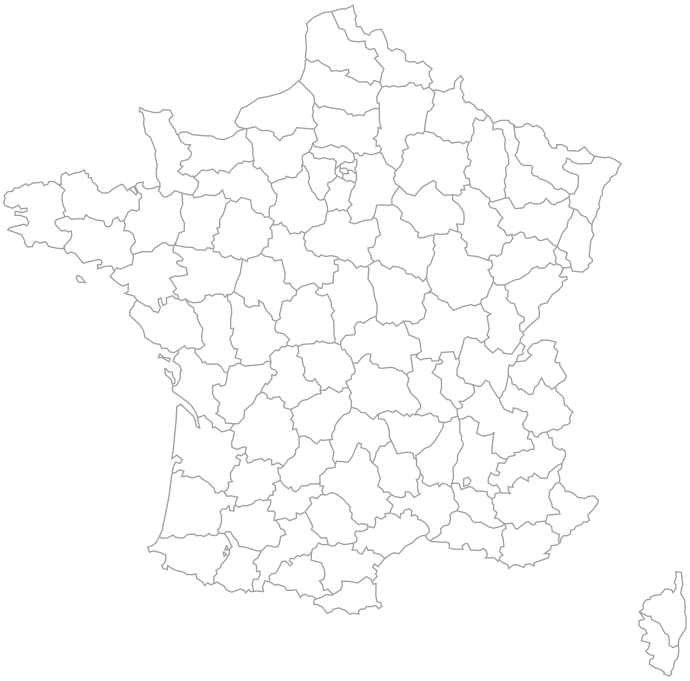


Рис. 13.17 Карта 96 департаментов Франции, составленная с использованием данных TopoJSON

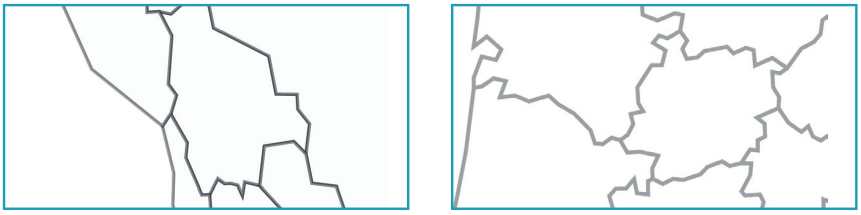
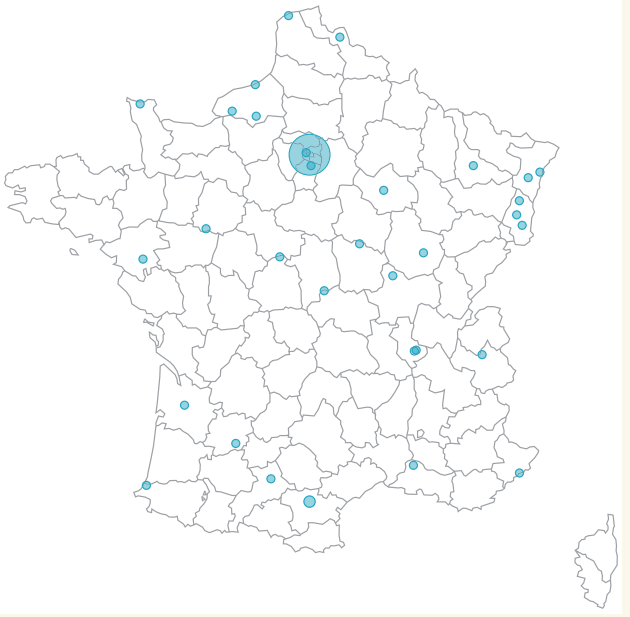


Рис. 13.18 В наборе данных TopoJSON каждая общая граница хранится как одна дуга, что позволяет создавать файлы меньшего размера и более аккуратные карты

Упражнение: визуализация французских городов, где родились лауреаты Нобелевской премии

Чтобы завершить проект, нарисуйте круги, представляющие французские города, где родились лауреаты Нобелевской премии. Для этого выполните те же шаги, что при рисовании городов на карте мира. Вот несколько рекомендаций:

1. отфильтруйте набор данных лауреатов, чтобы оставить только тех из них, кто родился во Франции;
2. извлеките даты и города рождения лауреатов, а также информацию, которая понадобится для рисования кругов;
3. добавьте круг для каждого города, где родился лауреат Нобелевской премии. Вычислите радиус с помощью функции `getCityRadius()` из `scales.js` и используйте широту и долготу городов для расчета координат окружностей в контейнере SVG;
4. по завершении ваша карта должна выглядеть, как показано на рисунке ниже.



Города на карте Франции, где родились лауреаты Нобелевской премии

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в разделе D.9 и в папке `13.5-TopoJSON/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

13.6 Дополнительные возможности

Как упоминалось выше, описание всех картографических возможностей D3 могло бы занять целую книгу. В этом разделе мы рассмотрим еще несколько возможностей, не упоминавшихся в данной главе.

13.6.1 Мозаичные карты

Мы можем создавать мозаичные карты, объединяющие отдельные блоки без швов подобно тому, как это сделано в Google Maps. Для наложения векторных объектов на растровые плитки можно использовать модули d3-geo и d3-tile. Если вы серьезно настроены на разработку мозаичных карт, то вам наверняка пригодятся специализированные библиотеки, такие как Mapbox GL (www.mapbox.com/mapbox-gl-js/api/).

13.6.2 Рисование на холсте

Функция `path.context` объекта `d3.geoPath()` позволяет вам легко рисовать векторную графику в элементе `<canvas>`. Это может значительно повысить производительность, поскольку при использовании холста не происходит добавления новых элементов в дерево DOM, а изображение создается путем изменения цвета пикселей, что требует меньших вычислительных ресурсов. Но, с другой стороны, с изображением на холсте нельзя взаимодействовать так же, как с элементами DOM.

13.6.3 Растровое проецирование

Для отображения спутниковой карты местности на одной из проекций можно использовать функцию растрового проецирования, предлагаемую модулем d3-geo. Пример, созданный Майком Бостоком (Mike Bostock), можно увидеть по адресу http://mng.bz/gvyn.

13.6.4 Шестиугольники

Модуль d3-hexbin позволяет накладывать на карты шестиугольники, как показано на рис. 13.19. Этот вариант визуализации будет интересен тем, у кого имеются количественные данные в точечной форме и требуется агрегировать их по площади.

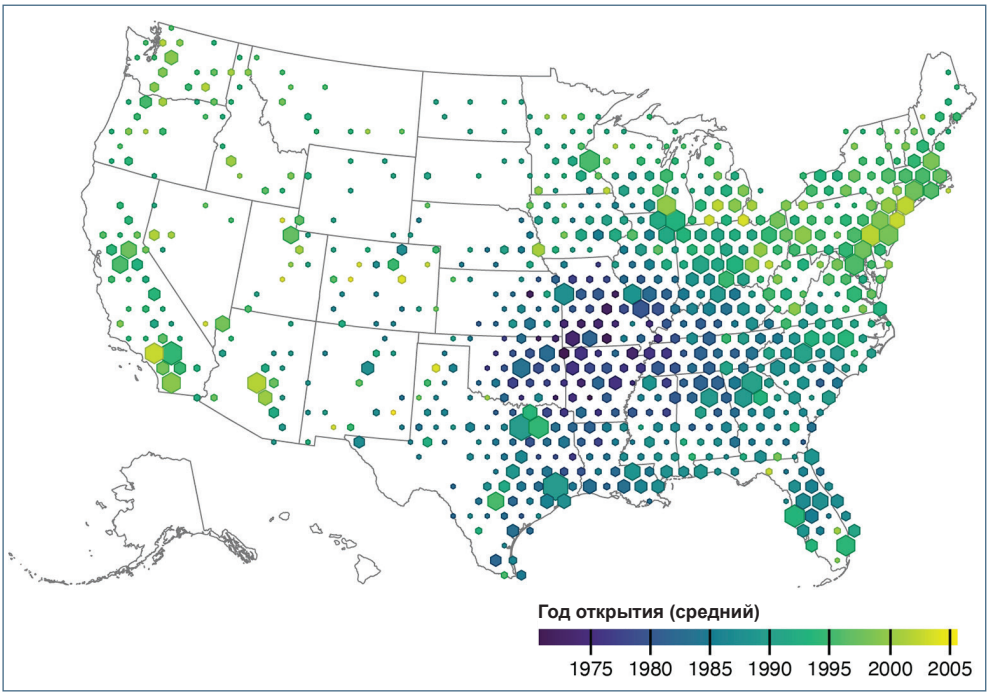


Рис. 13.19 Пример наложения шестиугольников на карту, созданный Майком Бостоком. Эта карта показывает размещение магазинов Walmart в США. Площадь каждого шестиугольника представляет количество магазинов в округе, а цвет – средний возраст этих магазинов (источник: https://observablehq.com/@d3/hexbin-map**)**

13.6.5 Диаграммы Вороного

Как и в случае с шестиугольниками, если у вас есть только точечные данные и вы хотите отобразить их на площади, то можете попробовать с помощью модуля d3-delaunay получить полигоны из точек, таких как столицы штатов (см. рис. 13.20). Диаграмма Вороного – это разбиение плоскости на ячейки, определяющее регионы, ближайшие к предопределенным точкам.

13.6.6 Карты-анаморфозы

Искажение площади или длины географического объекта для отображения другой информации создает карту-анаморфозу. Например, вы можете исказить улицы вашего города, исходя из времени, которое требуется, чтобы проехать по ним, или изменить размеры стран на карте мира в соответствии с численностью населения.

Простых функций для создания карт-анаморфоз в D3 не существует, однако можно использовать внешние библиотеки, такие как World Population Cartogram Мэтта Дзугана (Matt Dzugan; http://mng.bz/Y7rQ). Пример показан на рис. 13.21.

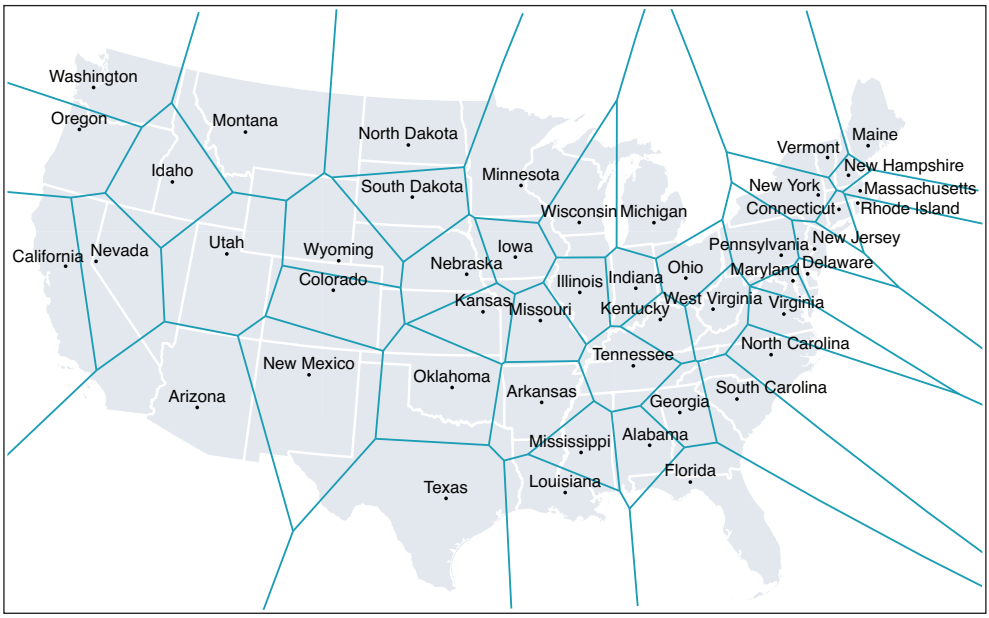


Рис. 13.20 Пример диаграммы Вороного, делящей территорию США на многоугольники по принципу близости к столице штата. Создана Майком Бостоком (источник: http://mng.bz/OZMO)

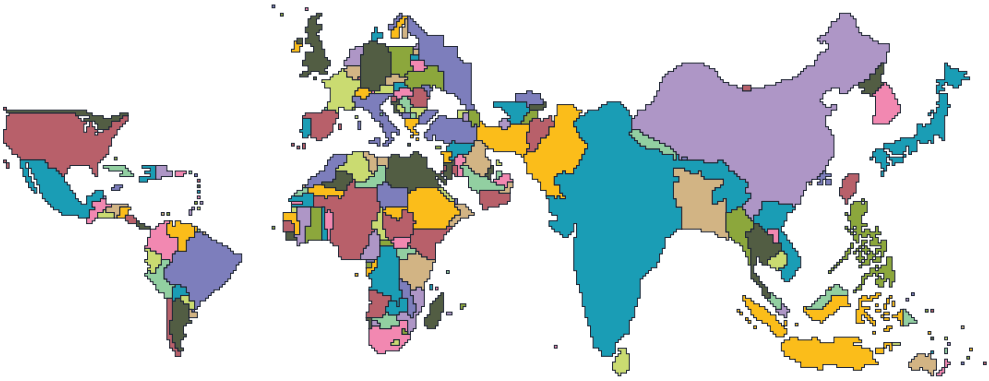


Рис. 13.21 Картограмма, созданная Мэттом Дзуганом, на которой размеры стран соответствуют численности их населения (источник: http://mng.bz/GZd8)

Интервью с Мартином Ламбрехтсом

Ламбрехтс – инженер по визуализации данных в компании Planet.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

Я – биоинженер, и в университете я прослушал факультативный курс по ГИС (географическим информационным системам). Именно тогда я научился делать карты. По окончании университета я сменил множество мест работы, но все их объединяло одно: мне приходилось сообщать цифры другим. И в этой области для меня, как и для многих других, настоящим открытием стали книги Эдварда Тафти (Edward Tufte). Из них я узнал, что создание визуализаций данных может не ограничиваться нажатием нескольких кнопок в Excel.

Проработав некоторое время в редакциях в роли дата-журналиста, где визуализация данных была значительной частью моей работы, я решил стать фрилансером, и именно этим я занимаюсь последние пару лет.

Как вы изучали D3? С какими трудностями столкнулись и как их преодолели?

Я вошел в сферу интерактивной визуализации данных как раз в ту пору, когда на смену Flash пришла D3. Я пытался овладеть ею, копируя и редактируя блоки (я пользовался платформой bl.ocks.org, созданной Майком Бостоком, создателем D3, на которой люди делились фрагментами кода), но настоящий прорыв случился, когда я повредил колено. Мне пришлось перенести операцию на колене, которая обездвижила меня на шесть недель, и я решил использовать это время для изучения веб-разработки и D3. В этот период я создал Weather Browser (www.maartenlambrechts.be/vis/weatherbrowser/), мой первый большой проект по визуализации данных, который я заявил на участие в конкурсе по визуализации данных. Он выиграл серебро, что вдохновило меня на продолжение.

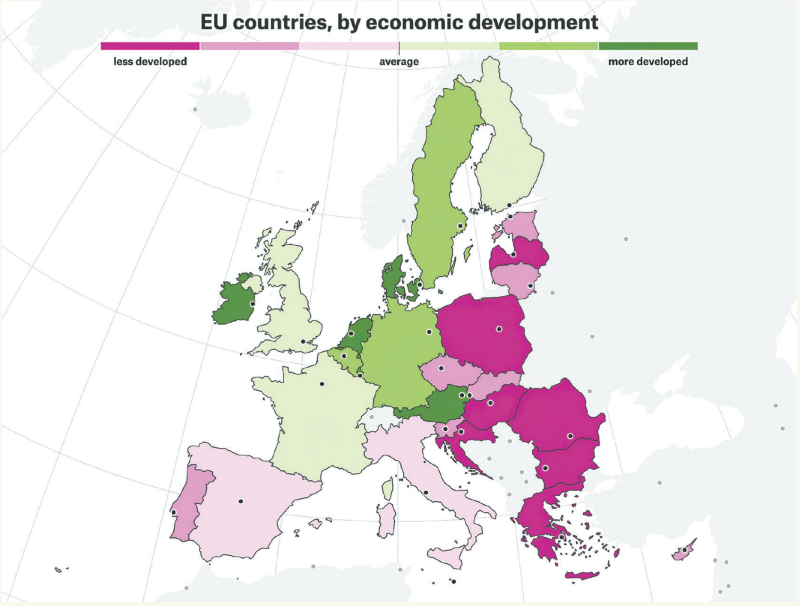
Моя главная проблема заключалась в отсутствии опыта программирования. Поэтому первое время я копировал много кода, но не знал, как он работает. В какой-то момент я осознал, что должен изучить основы JavaScript и прочитать документацию D3, чтобы четко понимать, как работает написанный мною код.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашем рабочем процессе визуализации данных?

Раньше я использовал все функции взаимодействия с DOM, имеющиеся в D3, но в настоящее время для этой цели я использую фреймворк Svelte, а из D3 применяю только вспомогательные функции, такие как масштабы, генераторы форм, извлечение данных и модуль d3-geo для создания карт.

Для обработки данных я использую пакеты tidyverse на R. Обычно я экспортирую CSV-файлы из R, которые затем загружаю в свои приложения D3.

Представленный проект: «Why Budapest, Warsaw, and Lithuania Split Themselves in Two» (https://pudding.cool/2019/04/eu-regions/)



Страны ЕС по уровню экономического развития

Опишите ваш проект «Why Budapest, Warsaw, and Lithuania Split Themselves in Two».

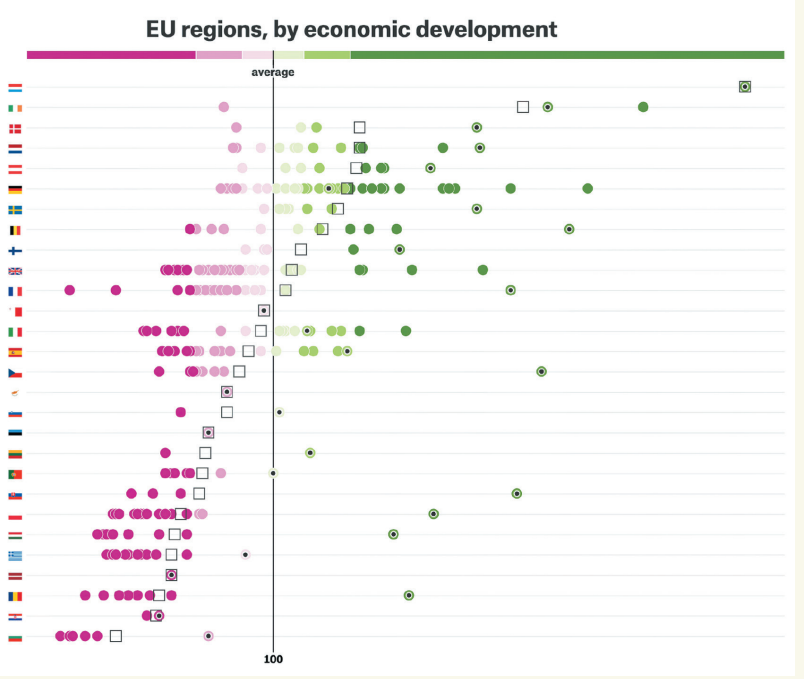
Первоначальная идея возникла у меня, когда я еще работал в редакции фламандской газеты «De Tijd». Но тогда у меня почти не было навыков программирования и мои редакторы не испытывали достаточного интереса, чтобы продолжить эту историю.

Тем не менее я создал базовый прототип, который пару лет спустя я представил в «The Pudding». Я посчитал, что способ перераспределения финансовых средств в ЕС между богатыми и бедными регионами плохо известен общественности, и попытался найти ракурс, подходящий для освещения в печати. Сначала я думал, что вижу некоторые признаки мошенничества: мне казалось, что регионы манипулируют своими показателями ВВП, чтобы оставаться ниже определенных пороговых значений. Но после разговора с некоторыми статистиками в Европейской комиссии я понял, что это не так. Поговорив с этими людьми, я узнал, что некоторые регионы перерисовывают свои границы, чтобы оптимизировать распределение средств в их пользу. И это была достаточно интересная история, которая очень хорошо поддавалась визуальному объяснению.

Расскажите, как создавался этот проект.

Я начал со сбора данных и создания некоторых начальных визуализаций с помощью ggplot2 в R. Так я увидел несколько подозрительных

пиков на некоторых гистограммах, которые, впрочем, не говорили о мошенничестве. Я продолжал обсуждать идею с Мэттом, моим редактором в «The Pudding» (https://pudding.cool/), пока мы не нашли правильный ракурс. Затем мы обсудили, как правильно преподнести данные и какие визуальные эффекты лучше всего использовать для этого.



Страны ЕС по уровню экономического развития

Для разработки я использовал шаблон, применявшийся разработчиками «The Pudding». В него были встроены все необходимые функции, такие как D3 и компонент scrollytelling. Затем несколько месяцев ушло на воплощение визуализации в код. Так много времени потребовалось просто потому, что параллельно я разрабатывал еще несколько проектов. Но в этом был свой плюс – мы с Мэттом смогли попробовать разные варианты и вывести визуализацию на новый уровень как по содержанию, так и по дизайну.

В этом проекте мне особенно нравится целенаправленность анимаций и переходов. Они, несомненно, поддерживают ваше видение ситуации. Вы с самого начала знали, какой тип повествования хотели создать, или это был итеративный процесс?

Основные анимационные эффекты и повествование уже были частью базового прототипа, созданного в прошлом. То есть анимация, которая переводит регионы на карте в точечный график и тем самым гарантирует понимание, что каждая точка на графике представляет регион,

который читатели только что видели на карте, была частью первоначальной идеи.

Давайте обсудим переход с изменением формы, который преобразует контуры стран в круги на полосовом графике. Какую библиотеку вы использовали для создания этого эффекта?

Трансформация регионов в круги реализована с помощью Flubber (https://github.com/veltman/flubber). Я познакомился с создателем Flubber, Ноа Вельтманом (Noah Veltman), на конференции OpenVis в 2018 году, где он представил свою библиотеку для интерполяции фигур SVG. В одном из примеров он показал преобразование контуров штатов США друг в друга, и я думаю, что именно этот пример меня вдохновил.

Что бы вы посоветовали нашим читателям, желающим заняться журналистикой данных?

Я всегда говорю, что главное – это практика. Вы можете прочитать много умных книг и статей о том, как что-то сделать, но по-настоящему овладеть мастерством можно, только делая это что-то своими руками. Начните с малого, примените прием, о котором только что узнали, к своим локальным данным и локальной среде.

Если же говорить более конкретно, то я настоятельно рекомендую изучить R и пакеты tidyverse, ggplot2 (для визуализации), dplyr и tidyr (для очистки и изменения формы данных). Они приучат вас задумываться о структуре ваших данных и о том, как их визуализировать.

Для изучения приемов интерактивной визуализации данных лучшим ресурсом является, пожалуй, Observable. Это интерактивная среда блокнотов, соучредителем которой является Майк Бостоk. Многие используют ее для экспериментов с визуализациями, и многие блокноты используют различные части D3.

Итоги

- Географические данные могут быть представлены в разных формах: от простых географических координат (широта и долгота) до наборов данных GeoJSON и TopoJSON.
- Основное различие между GeoJSON и TopoJSON состоит в том, что для каждого объекта, описывающего точку, линию или многоугольник, GeoJSON представляет массив координат, а TopoJSON хранит массив дуг.
- Данные GeoJSON легко создавать и читать, но файлы в этом формате могут быстро разрастаться в размерах.
- Данные TopoJSON позволяют отображать географические объекты с помощью файлов меньшего размера, что делает этот формат более подходящими для веб-проектов.

- Данные GeoJSON можно обрабатывать непосредственно с помощью D3, а для обработки TopoJSON требуется предварительное декодирование с применением внешней библиотеки.
- Под проекциями понимаются способы визуализации точек с трехмерного глобуса на плоской двумерной поверхности. Проекции неизбежно искажают географические данные в той или иной степени. D3 предоставляет методы, реализующие четыре вида проекций: азимутальную, коническую, цилиндрическую и составную.
- Для создания карты с помощью D3 требуется выполнить три основных шага: определить проекцию, инициализировать генератор элементов `path` с помощью `d3.geoPath()` и добавить эти элементы в контейнер SVG.
- Сетку меридианов и параллелей можно нарисовать с помощью функции `d3.geoGraticule()`.
- Чтобы нарисовать точки на карте, нужно передать их координаты в функцию проекции, которая вернет соответствующие координаты (`x`, `y`) в контейнере SVG.
- В D3 имеются модули, предназначенные для функций масштабирования и выделения.

Часть IV

Продвинутые методы визуализации

Теперь, попробовав создать множество разных визуализаций с помощью D3, мы перейдем к самой интересной части! В главе 14 мы создадим совершенно новый проект, объединив несколько модулей D3, и обсудим стратегию создания уникальной работы и формирования увлекательных повествований. Затем, в главе 15, мы поговорим о производительности и когда может понадобиться применить Canvas вместо SVG. Получив эти знания, вы будете готовы взяться за сложные проекты.

14 Создание нестандартных визуализаций

В этой главе рассматриваются:

- визуальное представление количественных и качественных данных;
- декомпозиция сложного проекта;
- объединение методов D3 для создания нестандартной визуализации;
- создание адаптивной сетки SVG;
- размещение визуальных элементов с использованием радиальной компоновки.

В первых трех частях этой книги мы применяли разные методы из D3 для разработки известных макетов визуализации, таких как столбчатые диаграммы, потоковые графики, гистограммы, карты и т. д. Но если вы выбрали D3 на роль инструмента визуализации данных, то высока вероятность, что вы также пожелаете создавать сложные и необычные визуализации. Для создания уникальных проектов важно знать, какие методы и макеты имеются в D3, и понимать, как они работают. Речь идет не столько о том, чтобы знать детали о каждом отдельном методе, сколько о том, чтобы освоить философию, лежащую в основе D3, и знать, где искать информацию, когда это необходимо. Приложение C, где представлены все модули

D3 с краткими описаниями, может вам помочь в этом. Еще один навык, который вам понадобится для создания нестандартных визуализаций макетов, – умение преобразовывать идеи и геометрические формы в код, что мы и сделаем в проекте, представленном в этой главе.

Этот проект даст вам возможность заглянуть за кулисы создания нестандартной визуализации, от эскизных идей до декомпозиции проекта на компоненты и отображения визуальных элементов в радиальном макете. Проект, который мы создадим здесь, исследует художественные произведения Винсента ван Гога, созданные им в последние десять лет его жизни. Завершенный проект вы найдете по адресу http://mng.bz/z8rg.

Мы будем следовать процессу из шести шагов. Это примерно та методология, которой вы можете следовать при создании любого проекта визуализации данных:

1. сбор данных;
2. изучение данных;
3. создание эскиза макета;
4. создание основы проекта;
5. создание визуализаций;
6. планирование осмысленных взаимодействий.

14.1 Сбор данных

Сбор и предварительная обработка данных – самый важный этап любого проекта визуализации. Если повезет, то можно получить уже готовые к использованию наборы данных и сразу начать работать над визуализацией, как в предыдущих проектах в этой книге. Но чаще всего требуется собрать данные из разных источников, проанализировать, обработать и отформатировать их. Сбор и обработка данных могут занять значительное время, что потребует от вас терпения и усердия. В этом разделе мы обсудим различные шаги, которые мы проделали, чтобы подготовить данные для проекта этой главы.

Но прежде чем начать искать данные, давайте определим, какую информацию мы будем визуализировать. Вдохновением для этого проекта стал семинар по визуализации данных с Фредерикой Фрагапане (Frederica Fragapane; www.behance.net/FedericaFragapane), в ходе которого мы работали с набором данных писем, написанных Винсентом Ван Гогом своему брату Тео. Мы были поражены богатством литературного наследия Ван Гога и подумали, что было бы интересно объединить его письма и картины, чтобы дать представление обо всем художественном наследии.

Итак, нам нужно собрать данные о картинах, рисунках и письмах. В идеале было бы желательно упорядочить их во времени, чтобы визуализировать взлеты и падения в его художественном творчестве. После нескольких сеансов поиска в Google мы нашли следующие ресурсы:

- *список работ Винсента Ван Гога* (https://en.wikipedia.org/wiki/List_of_works_by_Vincent_van_Gogh) – страница Википедии, где перечислены все картины, созданные Ван Гогом, а также наброски, которые он отправлял в письмах, сгруппированные по технике исполнения и периоду;
- *список рисунков Винсента Ван Гога* (https://en.wikipedia.org/wiki/List_of_drawings_by_Vincent_van_Gogh) – страница Википедии, где перечислены все рисунки, созданные Ван Гогом, за исключением его набросков в письмах;
- *Винсент Ван Гог: письма* (https://vangoghletters.org/vg/by_period.html) – сайт, посвященный письмам Ван Гога, сгруппированным по периодам;
- *Винсент Ван Гог* (https://en.wikipedia.org/wiki/Vincent_van_Gogh) – страница Википедии о жизни Ван Гога, которая может помочь нам лучше понять события, сформировавшие его художественное творчество.

Исследуя эти ресурсы, мы также заметили, что жизнь Ван Гога можно разбить на этапы по городам, в которых он жил. Например, он переехал из Нидерландов в Париж в 1886 году, где встретил Поля Гогена и Анри де Тулуз-Лотрека. Эти встречи, несомненно, повлияли на творчество Ван Гога. Мы также знаем, что он проходил лечение в приюте Сен-Поль-де-Мозоль с мая 1889 по май 1890 года. В этот период он начал включать завитки в свои изображения сада больницы. Наконец, Ван Гог покончил с собой в июле 1890 года, прервав 10-летие плодотворного художественного творчества. Зная об этих событиях, мы решили создать визуализацию в форме временной шкалы, включающей последние 10 лет жизни Ван Гога.

Теперь нам нужно извлечь данные из найденных ресурсов. Возьмем в качестве примера картины (http://mng.bz/0GZW). Эта страница Википедии содержит ряд таблиц, в которых перечислено чуть больше тысячи картин, но нам не хотелось бы извлекать все эти данные вручную! В интернете можно найти онлайн-сервисы, извлекающие таблицы из веб-страниц и преобразующие их в файлы CSV, например TableConvert (https://tableconvert.com). Это очень удобные и быстрые инструменты. Но если вам нужно более точно управлять извлечением, то для выполнения этой работы можно написать сценарий, это будет совсем несложно.

В листинге 14.1 показан сценарий, который можно использовать для извлечения названия, URL изображения и техники, в которой написана каждая картина, представленная в списке на странице в Википедии. Чтобы использовать этот сценарий, скопируйте его в буфер обмена, откройте консоль браузера, выполните вставку из буфера обмена и нажмите **Enter**.

Посмотрев на структуру страницы, вы увидите, что она состоит из набора таблиц HTML, каждая из которых содержит список картин, выполненных в одной и той же технике. Первые шесть таблиц посвящены живописи маслом; седьмая – акварели; восьмая и девятая – литографиям и офортам, которые мы сгруппируем в технику «print» (печать). Последняя таблица содержит наброски, нарисованные в письмах, которые мы пока не будем извлекать. В листинге 14.1 мы объявляем массив, содержащий индекс интересующих нас таблиц и связанных с ними техник исполнения.

Затем мы извлекаем все таблицы HTML из страницы, используя метод документа `querySelectorAll()` и классы `"wikitable"` и `"sortable"` в качестве селектора. Мы выяснили этот селектор, открыв инспектор браузера и внимательно изучив разметку, когда пытались определить уникальный селектор для интересующих нас таблиц.

Выполняя циклический обход этих таблиц, мы проверяем, существуют ли они уже в массиве `tables`, объявленном в начале скрипта. Эта проверка помогает нам миновать таблицы с набросками в письмах. Затем мы просматриваем каждую строку в таблице и извлекаем название и URL изображения картины. Обратите внимание, что нам пришлось разместить в коде различные структуры DOM, потому что строки таблицы не всегда отформатированы одинаково. Названиям и URL картин, которые не соответствуют этим структурам HTML, присваивается значение `null` – они будут добавлены вручную позже. Работа с реальными данными часто бывает запутанной! Также обратите внимание, что мы извлекаем URL изображения из атрибута `srcset` вместо `src`, потому что изображения, на которые ссылается этот атрибут, меньше и их загрузка в проект займет меньше времени.

Наконец, мы помещаем информацию о картине в объект и добавляем его в массив с именем `paintings`. Если по окончании вывести содержимое этого массива в консоль, то его можно скопировать и вставить в текстовый редактор и создать файл JSON.

Этот сценарий написан под данный конкретный пример и будет бесполезен на другой веб-странице. Но он ясно показывает, насколько ценными могут быть навыки программирования на JavaScript для извлечения информации из веб-страницы.

Листинг 14.1 Извлечение данных о картинах Ван Гога из Википедии

```
const tables = [
  { index: 0, medium: "oil" },
  { index: 1, medium: "oil" },
  { index: 2, medium: "oil" },
  { index: 3, medium: "oil" },
  { index: 4, medium: "oil" },
  { index: 5, medium: "oil" },
  { index: 6, medium: "watercolor" },
  { index: 7, medium: "print" },
  { index: 8, medium: "print" },
];
```

Создать список таблиц для извлечения информации

Инициализировать пустой массив и сохранить его в константе с именем paintings. Сюда будет сохраняться информация о каждой картине

```
const domTables = document.querySelectorAll(
  ".wikitable.sortable"
);
const paintings = [];
```

Выбрать все таблицы HTML с классом "sortable"

```
domTables.forEach((table, i) => {
  if (i <= tables.length - 1) {
    const medium = tables[i].medium;

    const rows = table.querySelectorAll("tbody tr");
    rows.forEach(row => {
      let title;
      if (row.querySelector(".thumbcaption i a")) {
        title = row.querySelector(".thumbcaption i a")
          .textContent;
      } else if (row.querySelector(".thumbcaption i")) {
        title = row.querySelector(".thumbcaption i")
          .textContent;
      } else {
        title = null;
      }

      let imageLink;
      if (row.querySelector(".thumbinner img")) {
        const image = row.querySelector(
          ".thumbinner img"
        ).srcset;
        imageLink = `https${image.slice(
          image.indexOf("1.5x, ")+6,-3
        )}`;
      } else {
        imageLink = null;
      }
```

Выполнить цикл по выбранным таблицам. Если таблица присутствует в массиве tables, то сохранить название соответствующей техники в константе и выполнить цикл по строкам таблицы

Извлечь название картины и сохранить его в константе

Извлечь URL изображения картины и сохранить его в константе

```
      paintings.push({
        title: title,
        imageLink: imageLink,
        medium: medium
      });
    })
  }
});

console.log(paintings);
```

Добавить название, URL изображения и технику исполнения в объект и сохранить объект в массив paintings

Вывести массив картин в консоль

Пример сценария в листинге 14.1 неполный. Нам еще нужно извлечь дату, размер и место создания каждой картины. Мы не будем делать этого здесь, чтобы не увеличивать размер раздела, но вы можете попробовать попрактиковаться в извлечении данных с веб-страницы! Обратите внимание, что нам также пришлось пойти на некоторые ухищрения, чтобы сохранить отдельно ширину и высоту картин, а также месяц и год создания. Нам понадобилось провести некоторые дополнительные исследования, чтобы узнать месяц создания некоторых картин и найти их жанр (портрет, натюрморт, пейзаж и т. д.).

Если вы хотите сразу перейти к использованию данных, то файлы с готовыми наборами, содержащие информацию о картинах, набросках и письмах Ван Гога, вы найдете в репозитории главы (http://mng.bz/KZ5E).

14.2 Изучение данных

В главе 3 мы определили две основные категории данных: *количественные* и *качественные* (показаны на рис. 14.1). Количественные данные состоят из числовой информации, такой как стоимость акций на фондовом рынке или количество студентов в классе. Количественные данные могут быть *дискретными*, т. е. состоящими из целых чисел, которые нельзя разделить, или *непрерывными*, где числа остаются значимыми при делении на более мелкие единицы. Качественные данные представляют нечисловую информацию, такую как список стран или доступные объемы кофе для заказа в Starbucks (шорт, толл, гранде, венти и т. д.). Качественные данные могут быть *номинальными*, значения которых не имеют определенного порядка, или *порядковыми*, для которых порядок имеет значение.



Рис. 14.1 Данные можно сгруппировать в две основные категории: количественные (числовая информация) и качественные (нечисловая информация). Количественные данные могут быть дискретными или непрерывными, а качественные – номинальными или порядковыми

Поскольку мы будем использовать разные каналы для визуализации разных типов данных, часто бывает полезно составить список переменных, которые есть в нашем распоряжении, и организовать их по типам данных. Этот шаг может помочь определить различные способы визуального представления данных, которые мы можем использовать. Как показано на рис. 14.2, количественные данные часто визуализируются с помощью *местоположения* (как в диаграмме рассеяния), *длины* (как в столбчатой диаграмме), *площади* (как узлы в нашем проекте «Ромео и Джульетта» (см. главу 12)), *углов* (как в круговой диаграмме) или с помощью *непрерывной цветовой шкалы*. Качественные данные, с другой стороны, часто представляются с помощью *категориальной цветовой шкалы*, *узоров*, *символов*, *связей*, как в сетевом графике, или *вложения* иерархических данных, как в упакованных пузырьковых диаграммах. Это далеко неполный список, потому что, применив творческий подход, можно придумать новые способы визуализации данных, но он помогает получить представление об основных способах визуального кодирования информации, имеющихся в нашем распоряжении.



Рис. 14.2 Количественные данные часто визуализируются с помощью местоположения, длины, площади, угла и непрерывной цветовой шкалы, а качественные данные – с помощью категориальной цветовой шкалы, узоров, символов, связей и вложения

На данном этапе полезно перечислить различные атрибуты имеющихся у нас данных, классифицировать количественные и качественные данные и подумать о том, как лучше визуализировать основные атрибуты. На рис. 14.3 перечислены имеющиеся у нас четыре набора данных (список картин Ван Гога, список рисунков, количество писем, которые он писал каждый месяц, и временная шкала с городами, в которых он жил в течение выбранного периода времени) и определяется, к какой категории данных они относятся – количественных (синие точки) или качественных (красные точки). Основываясь на этой информации, мы можем начать думать о визуализации, которую предполагается создать.

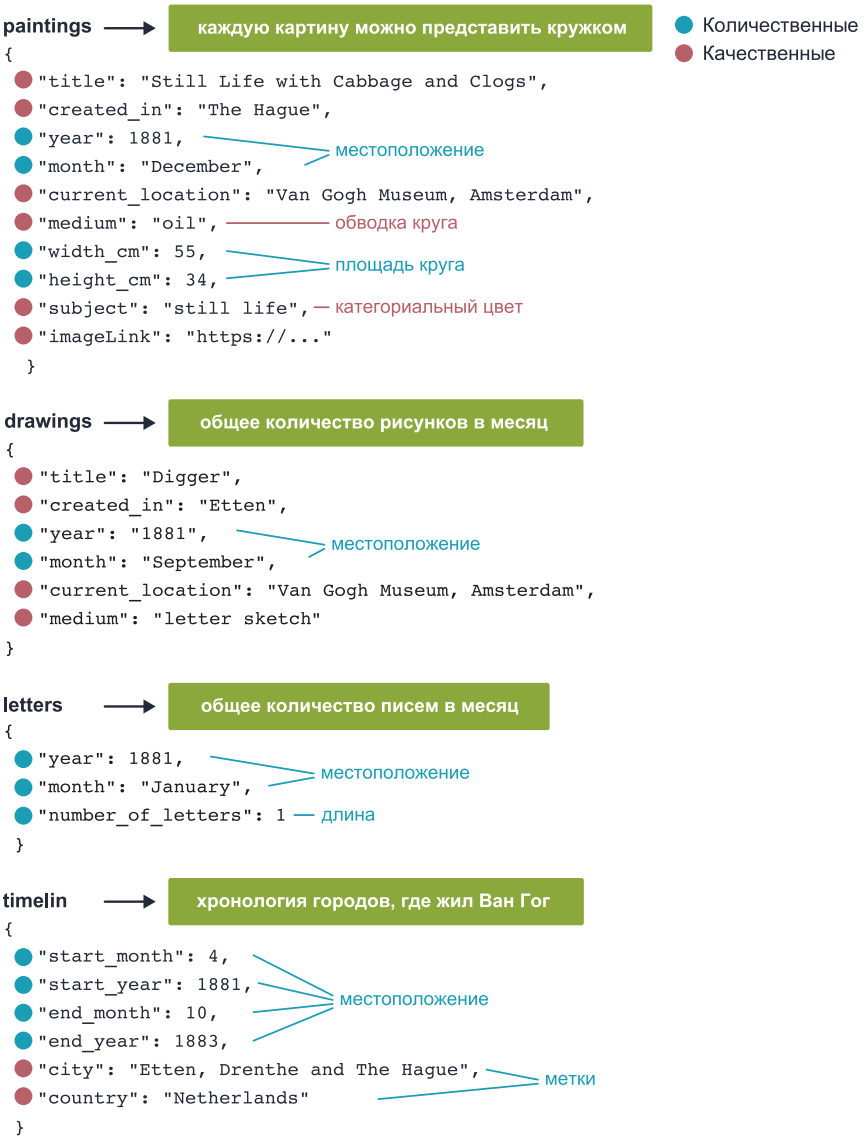


Рис. 14.3 Перед созданием визуализации полезно перечислить все атрибуты имеющихся данных, как количественные (синие точки), так и качественные (красные точки). Затем можно начать думать о том, как их визуализировать

В этом проекте мы хотим представить художественную продукцию Ван Гога (картины, рисунки и письма) на временной шкале, чтобы исследовать взаимосвязь между использованием каждого способа выражения и как они развивались по мере переездов художника по Нидерландам и Франции. Мы хотим больше сосредоточиться на картинах и позволить пользователям исследовать их по отдельности. Если представить каждую картину кругом, то мы могли

бы использовать цвет для сообщения жанра картины (портрет, натюрморт, пейзаж и т. д.), размеры произведения с помощью размера круга, а технику (масло, акварель или печать) с помощью обводки, как показано на рис. 14.4. Эти круги будут расположены на некоторой временной шкале.

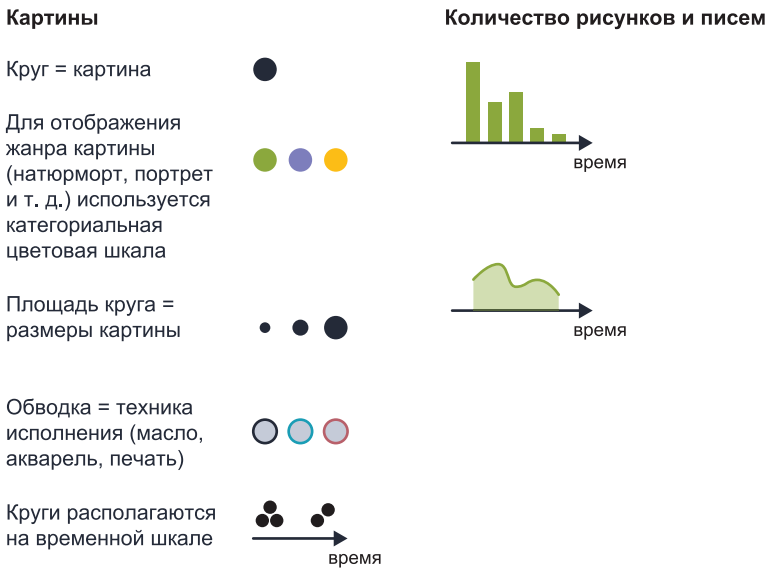


Рис. 14.4 После изучения атрибутов данных и выяснения их типов мы решили представить каждую картину кругом, расположенным на временной шкале. Цвет, размер и обводка круга будут сообщать о жанре, размерах и технике картины. Количество рисунков и писем мы визуализируем с помощью столбчатой диаграммы или диаграммы с областями, также расположенной на временной шкале

Количество рисунков и писем, созданных каждый месяц, можно добавить в качестве вторичной информации с помощью столбчатой диаграммы или диаграммы с областями. Наконец, мы знаем, что временная шкала должна быть интерактивной и позволять выбирать произведения Ван Гога, созданные им в разные периоды его жизни.

14.3 Создание эскиза макета

После выбора способов визуализации можно приступать к созданию макета проекта. Мы уже выяснили, что каждая картина будет представлена кругом и размещена на оси времени. Мы можем использовать горизонтальную или вертикальную ось, хотя они могут оказаться слишком большими для экрана. Интересным решением выглядит радиальная ось времени. И вместо одного большого круга,

который трудно адаптировать к мобильным экранам, можно было бы использовать подход с малыми кратными, состоящими из серии визуализаций, использующих тот же масштаб и оси, но представляющих другой аспект данных. При таком подходе можно для каждого года создать отдельную радиальную диаграмму, что позволит разместить их в сетке, как на рис. 14.5. На визуализации для настольно-

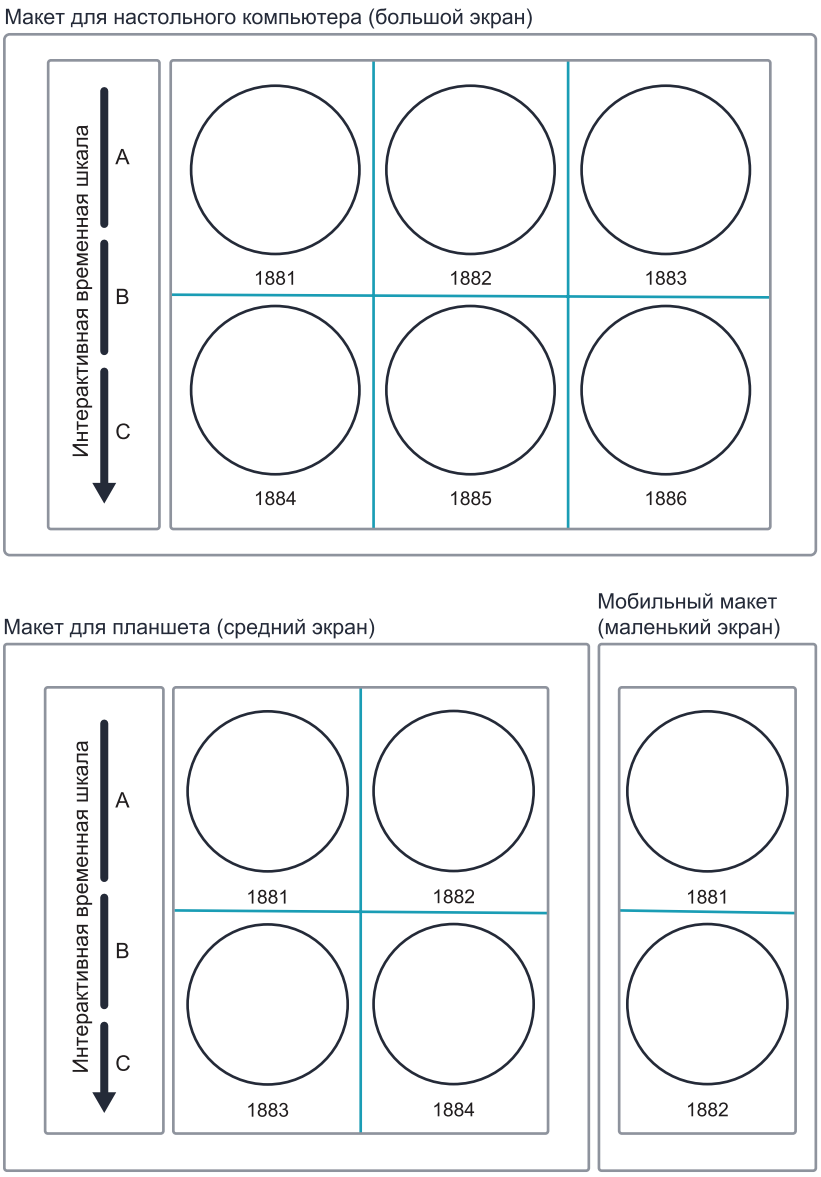


Рис. 14.5 Визуализации малых кратных состоят из серии диаграмм, имеющих одинаковый масштаб и оси. Обычно они помещаются в сетку, что позволяет легко адаптировать их к разным размерам экрана

го компьютера будет интерактивная временна́я шкала слева и сетка с тремя колонками справа. На планшетах сетка будет содержать только две колонки, а на мобильных устройствах – одну колонку без функциональности, характерной для временной шкалы.

Каждое малое кратное будет визуализировать целый год, а месяцы будут распределены по окружности круга. Количество рисунков, нарисованных в каждом месяце, будет представлено диаграммой с областями, а количество писем – длиной полосы. Круги, представляющие картины, написанные в течение месяца, будут сгруппированы вместе, как показано на рис. 14.6.

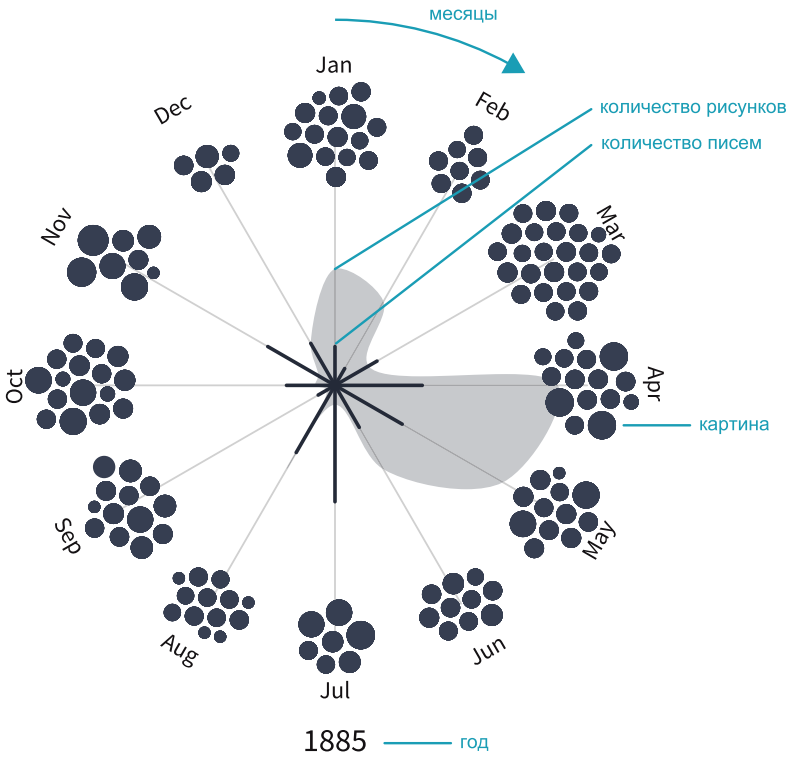


Рис. 14.6 Месяцы располагаются по кругу в каждой отдельной диаграмме, представляющей год. Количество рисунков, нарисованных в каждом месяце, будет представлено диаграммой с областями, а количество писем – длиной полосы. Картины, написанные в течение месяца, будут сгруппированы вместе

Следующий шаг – создание цветовой палитры и выбор шрифтов. Нам понадобится цветовая палитра для восьми различных жанров живописи – автопортрет, портрет, крестьянская жизнь, интерьер, натюрморт, пейзаж, городской пейзаж и все остальное – и еще один цвет для писем и рисунков. При создании любой цветовой палитры подумайте о том, какую атмосферу вы хотите создать. Например,

здесь мы хотели бы использовать жизнерадостную палитру с оттенками, характерными для картин Ван Гога последних нескольких лет его жизни. Мы создали категориальную цветовую палитру (см. рис. 14.7), извлекая основные цвета из картин и используя веб-сайт Coolors (https://coolors.co) для создания соответствующих цветов. Восемь цветов – это уже много для категориальной палитры, поэтому для некоторых категорий нам пришлось использовать похожие оттенки. Например, мы выбрали цвет увядшей розы для портретов (#c16e70) и более яркую версию того же цвета для автопортретов (#f7a3a6). Также вдохновение для цветовых палитр можно найти в Adobe Color (https://color.adobe.com) и Color Hunt (https://colorhunt.co).



Рис. 14.7 Для этого проекта нам понадобится категориальная палитра с восемью цветами для объектов живописи и еще одним цветом для писем и рисунков. Поэкспериментировав с яркостью, мы создали два варианта последнего цвета, которые будем использовать для текста и фона проекта

Отличным ресурсом для выбора бесплатных веб-шрифтов мы считаем Google Fonts (https://fonts.google.com). Старайтесь не слишком разнообразить применяемые шрифты, достаточно двух семейств шрифтов на проект – одно для заголовков и одно для основного текста. Простой поиск в Google даст вам множество идей для комбинирования шрифтов Google. Для этого проекта мы выбрали для заголовков шрифт с засечками Libre Baskerville Bold, который напоминает о XIX веке, и для текста и надписей – шрифт без засечек Source Sans Pro, который отлично читается в пользовательских интерфейсах.

14.4 Создание основы проекта

Определив общую конструкцию будущей визуализации, мы должны выбрать инфраструктуру. Поскольку этот проект сложнее тех, что демонстрировались ранее в нашей книге, мы используем фреймворк JavaScript. Это позволит нам разбить проект на небольшие компоненты, что упростит его разработку и поддержку. У нас уже есть опыт использования React в проекте в главе 8, поэтому на этот раз выберем фреймворк Svelte, который особенно нравится сообществу визуализации данных. Не волнуйтесь, если вы еще незнакомы с Svelte, потому что основное внимание будет уделено общим принципам создания сложного проекта. Вы можете читать и собирать знания по крупицам, не погружаясь в Svelte. Если у вас есть опыт использования Svelte или вам интересно попробовать его, то вы увидите, что он довольно интуитивно понятен и отлично работает с D3. Краткое введение в Svelte вы найдете в приложении E, удобные краткие руководства – по адресу https://svelte.dev/tutorial.

Объединяя D3 с фреймворком JavaScript или компилятором, таким как Svelte, мы должны разделить обязанности между ними. В нашем проекте фреймворк будет отвечать за добавление, удаление и изменение элементов DOM, а D3 – за выполнение вычислений, связанных с масштабами, генераторами форм, силовой компоновкой и т. д. Проще говоря, нам нужно забыть о шаблоне привязки данных и с осторожностью использовать переходы D3, чтобы избежать конфликтов между D3 и Svelte. Более подробное обсуждение возможных подходов к объединению D3 с фреймворком вы найдете в главе 8.

Чтобы начать работу над проектом, скачайте код из репозитория (http://mng.bz/9dA0) и откройте папку `start` в редакторе. Откройте также окно терминала и установите зависимости проекта командой `npm install`. Затем запустите проект с помощью `npm run dev`. Проект будет доступен в браузере по адресу `http://localhost:5173/`. Все файлы, с которыми мы будем работать, вы найдете внутри папки `src/`:

- `App.svelte` – корневой компонент нашего проекта, вызывающий дочерние компоненты, которые создают макет страницы;
- папка `/data` содержит четыре набора данных, которые будут использоваться в этом проекте: `drawings.json`, `letters.json`, `painting.json` и `timeline.json`;
- `global_styles/` содержит четыре файла Sass CSS (SCSS), которые подключаются в `styles.scss`. Использование SCSS вместо CSS позволяет объявлять переменные стиля (например, `$text: #160E13`), которые можем повторно использовать в проекте, что упрощает его поддержку;

- `layout/` содержит основные компоненты макета проекта, такие как заголовок и легенда;
- `chart_component/` содержит компоненты, относящиеся к определенным компонентам визуализации, таким как картины или письма;
- `/UI` содержит компоненты интерфейса, такие как всплывающая подсказка;
- `utils/` содержит служебные функции и константы.

14.4.1 Другой подход к созданию адаптивных контейнеров SVG

Из главы 1 мы узнали, как просто и эффективно сделать графику SVG адаптивной: для этого достаточно добавить в контейнер SVG атрибут `viewBox` и оставить атрибуты `width` и `height` пустыми. Этот подход очень прост в реализации и не требует дополнительных настроек. Единственный реальный недостаток – при уменьшении контейнера SVG вместе с ним пропорционально уменьшаются содержащиеся в нем текстовые элементы, что может сделать их трудночитаемыми.

В этом проекте мы используем другой подход: установим атрибуты `width` и `height` контейнера SVG и оставим атрибут `viewBox`. Для обновления значений этих атрибутов мы используем обработчик события изменения размеров экрана. Этот подход потребует от нас приложить больше усилий, зато позволит адаптировать макет визуализации с учетом ширины экрана. Кроме того, он сохранит размер текстовых меток при уменьшении размера экрана.

В предшествующем обсуждении мы решили отобразить сетку из нескольких визуализаций. Все эти визуализации будут заключены в один элемент SVG. Кроме того, для общей разметки страницы, включающей временную шкалу и визуализации, мы используем 12-колоночную сетку flexbox, похожую на ту, что обсуждалась в главе 9.

На рис. 14.8 показан макет страницы при трех различных значениях ширины экрана: от 1400 пикселей и больше (ширина контейнера сетки flexbox), от 748 до 1400 пикселей и меньше 748 пикселей. Для каждого из этих трех размеров экрана ширина контейнера SVG вычисляется по-своему. Когда ширина экрана больше или равна 748 пикселям, то временная шкала отображается слева, занимая 2 колонки из 12, а визуализация (контейнер SVG) – справа, в 10 оставшихся колонках. Когда экран меньше 748 пикселей, временная шкала не отображается, и визуализация может занять все 12 колонок. Контейнер сетки flexbox также имеет отступы слева и справа по 30 пикселей.

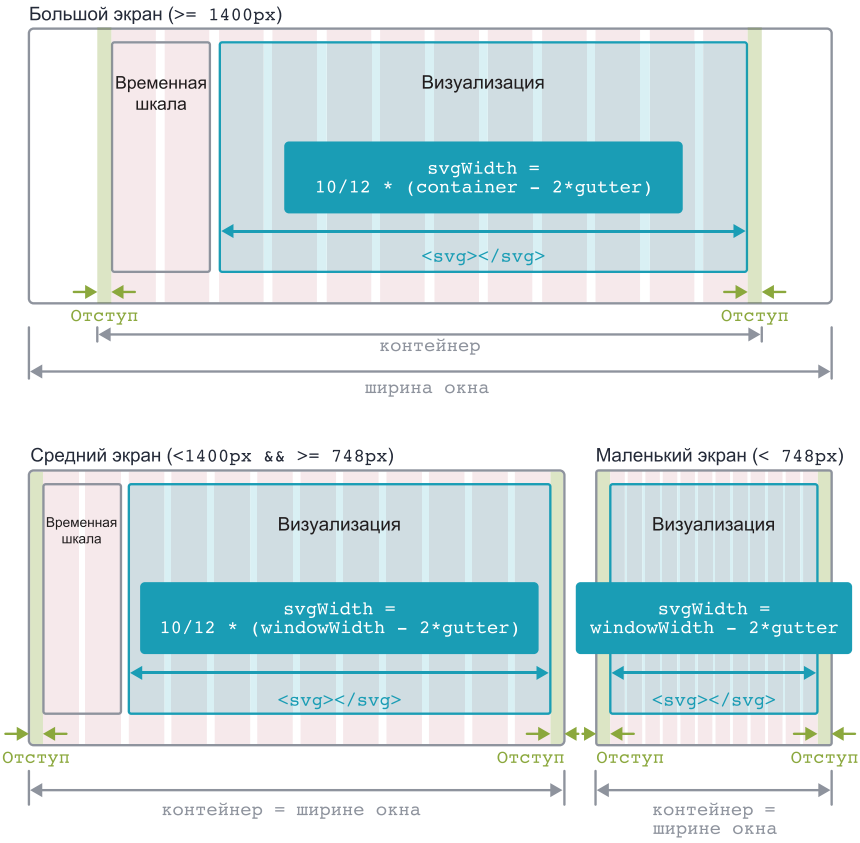


Рис. 14.8 Ширина контейнера SVG пропорциональна ширине экрана и адаптивной компоновке страницы. На экранах с шириной больше 748 пикселей контейнер SVG занимает 10 столбцов в 12-колоночной сетке flexbox, а на экранах меньшего размера – все 12 столбцов

Сетка flexbox ограничена максимальной шириной в 1400 пикселей. Это означает, что даже на больших экранах содержимое не превысит эту ширину. Чтобы вычислить ширину контейнера SVG на экранах шире 1400 пикселей, можно вычесть отступы с каждой стороны контейнера сетки flexbox, умножить полученную разность на 10, а затем разделить на 12. Это объясняется тем, что контейнер SVG занимает 10 из 12 столбцов:

```
svgWidth = 10 ÷ 12 × (gridContainer - 2 × отступ)
```

Когда ширина экрана меньше 1400 пикселей, размер контейнера SVG пропорционально уменьшается. В уравнении вычисления `svgWidth` нужно лишь заменить `gridContainer` на `windowWidth`:

```
svgWidth = 10 ÷ 12 × (windowWidth - 2 × отступ)
```

Наконец, когда ширина экрана меньше 768 пикселей, контейнер SVG занимает всю его ширину за вычетом отступов:

```
svgWidth = windowWidth - 2 × отступ
```

Используем эти уравнения для динамического вычисления ширины контейнера (см. листинг 14.2). Для этого откройте файл `Grid.svelte`. Сначала объявим две переменные, `windowWidth` и `gridContainer`. С помощью оператора `switch` установим значение переменной `svgWidth`, исходя из ширины экрана и используя уравнения, которые мы только что обсудили. Поскольку оператор `switch` объявлен со знаком реактивности Svelte (`$`), он будет выполняться при каждом изменении содержащейся в нем переменной.

Обратите внимание, что мы привязали переменную `windowWidth` к свойству `innerWidth` объекта `window`. В Svelte можно получить доступ к объекту `window` из любого компонента с помощью `<svelte:window />`.

Наконец, динамически установим атрибут `width` контейнера SVG с помощью переменной `svgWidth`. При работе с фреймворком JavaScript не принято добавлять элементы в DOM с помощью D3, поэтому мы добавляем их непосредственно в разметку компонента.

Листинг 14.2 Динамическое обновление ширины (Grid.svelte)

```
<script>
   let windowWidth;
   const gridContainer = 1400;
   const padding = 30;
   let svgWidth;
   $: switch (true) {
       case windowWidth >= gridContainer:
         svgWidth = (10 / 12) * (gridContainer -
           2 * padding);
         break;
       case windowWidth < gridContainer &&
         windowWidth >= 768:
         svgWidth = (10 / 12) * (windowWidth -
           2 * padding);
         break;
       default:
         svgWidth = windowWidth - 2 * padding;
     }
</script>

<svelte:window bind:innerWidth={windowWidth} />
```

При каждом изменении переменной windowWidth вычисляется svgWidth. Это возможно благодаря реактивному блоку, созданному со знаком $

Привязать значение переменной windowWidth к свойству innerWidth объекта window. Объект window доступен внутри любого компонента Svelte!

```
<svg width={svgWidth} />

<style>
  svg {
    border: 1px solid magenta;
  }
</style>
```

Динамически присвоить атрибуту width контейнера SVG значение svgWidth

В разделе "styles" в элемент SVG добавлена пурпурная рамка. Вы можете поэкспериментировать с изменением размера экрана, чтобы увидеть, как это изменение влияет на ширину элемента SVG.

Теперь, решив вопрос с установкой ширины контейнера SVG, нам нужно задать его высоту. Поскольку элемент SVG будет содержать сетку из множества небольших визуализаций, мы можем вычислить его высоту, зная следующее: количество визуализаций, их высоту и количество столбцов в сетке. В листинге 14.3 мы начинаем с объявления массива годов, участвующих в визуализации. Мы делаем это с помощью метода `range` из библиотеки D3. Затем задается количество колонок в сетке, занимаемых визуализацией, на основе ширины экрана. Если ширина экрана больше 900 пикселей, то используются три колонки; если меньше 600 пикселей, то используется одна колонка, а при ширине от 600 до 900 пикселей – две колонки. Пока мы применяем приблизительные цифры и скорректируем их позже, если потребуется.

Определив количество колонок, мы можем вычислить количество строк, разделив количество элементов небольших визуализаций на количество колонок и округлив результат. Ширина каждого из элементов находится путем деления ширины элемента SVG на количество колонок. Мы также произвольно устанавливаем их высоту на их ширину плюс 40 пикселей. Наконец, находим общую высоту элемента SVG, умножая количество строк на высоту каждого элемента.

Поскольку при монтировании компонента переменным `svgWidth` и `svgHeight` присваивается значение `null`, браузер выдаст ошибку. Вот почему мы используем условный оператор для добавления элемента SVG в разметку только после определения этих двух переменных. Обратите внимание, что оператор `switch` и переменные размеров объявляются реактивными с помощью символа `$`. Они будут обновляться при каждом изменении ширины экрана.

Теперь у нас есть адаптивный элемент SVG! Для его реализации потребовалось приложить больше усилий, по сравнению с предыдущей стратегией, но этот новый подход поможет нам в следующем разделе, когда мы будем экспериментировать с адаптивной сеткой SVG.

Листинг 14.3 Динамическое обновление высоты (Grid.svelte)

```
<script>
  import { range } from "d3-array";

  ...

  const years = range(1881, 1891);
  let numColumns;
  $: switch (true) {
    case windowWidth > 900:
      numColumns = 3;
      break;
    case windowWidth <= 900 && windowWidth > 600:
      numColumns = 2;
      break;
    default:
      numColumns = 1;
  }
  $: numRows = Math.ceil(years.length / numColumns);
  $: smWidth = svgWidth / numColumns;
  $: smHeight = smWidth + 40;
  $: svgHeight = numRows * smHeight;
</script>

<svelte:window bind:innerWidth={windowWidth} />

{#if svgWidth && svgHeight}
     <svg width={svgWidth} height={svgHeight} />
{/if}

<style>
  svg {
    border: 1px solid magenta;
  }
</style>
```

Создать массив годов, которые будут участвовать в визуализации

Установить количество колонок сетки SVG на основе ширины экрана

Рассчитать количество строк сетки, разделив количество визуализаций на количество колонок

Рассчитать ширину и высоту каждого элемента сетки

Рассчитать высоту контейнера SVG, умножив количество строк на высоту каждого элемента сетки

Установить атрибут `height` элемента SVG. Добавить элемент SVG в разметку, только если определены обе переменные: svgWidth и svgHeight

14.4.2 Создание адаптивной сетки SVG

В листинге 14.3 мы задали ширину и высоту каждого элемента сетки с помощью переменных `smWidth` и `smHeight`. Используя эти значения, мы построим сетку, которая будет содержать все визуализации. Работая в контейнере SVG, мы будем использовать группы и заключать в них элементы визуализаций.

Для начала, как показано в листинге 14.4, мы вставляем блок `each` в контейнер SVG, который перебирает ранее созданный массив `years`. Стоит отметить, что, обратившись ко второму аргументу (`i`), можно получить доступ к индексу каждого года. Мы создаем элемент группы для каждого года, а затем используем атрибут `transform` для применения смещения. Чтобы определить, к какой колонке принад-

лежит каждая группа, мы используем остаток от деления индекса на количество столбцов (следующие уравнения иллюстрируют остаток от деления для индексов от нуля до пяти в компоновке с тремя колонками), а затем вычисляем горизонтальное смещение, умножая остаток на `smWidth`:

```
0 % 3 = 0
1 % 3 = 1
2 % 3 = 2
3 % 3 = 0
4 % 3 = 1
5 % 3 = 2
. . .
```

Для вычисления смещения по вертикали мы округляем результат деления индекса на количество столбцов, чтобы узнать, в какой строке находимся, и умножаем округленное значение на высоту элемента сетки. Затем добавляем элемент прямоугольника внутрь группы, устанавливаем его размеры равными ширине и высоте элемента сетки и добавляем синюю рамку. Прямоугольники нужны нам, чтобы убедиться, что сетка работает в соответствии с нашими ожиданиями и правильно меняет размеры при изменении ширины экрана, но мы уберем их из окончательной визуализации.

Листинг 14.4 Добавление адаптивной сетки (Grid.svelte)

```
{#if svgWidth && svgHeight}
  <svg width={svgWidth} height={svgHeight}>
    {#each years as year, i}
      <g transform="translate(
        {(i % numColumns) * smWidth},
        {Math.floor(i / numColumns) * smHeight})"
      >
        <rect x={0} y={0} width={smWidth}
          height={smHeight} />
      </g>
    {/each}
  </svg>
{/if}

<style>
  svg {
    border: 1px solid magenta;
  }
  rect {
    fill: none;
    stroke: cyan;
  }
</style>
```

Добавить блок each для перебора элементов массива years

Добавить элемент группы для каждого года и переместить его в нужное местоположение в сетке

Добавить прямоугольник в группу и задайть его размеры с помощью smWidth и smHeight. Этот элемент позволит нам убедиться, что сетка работает в точности как ожидалось. Мы уберем эти прямоугольники, как только визуализация будет готова

После реализации сетки попробуйте изменить размер экрана и убедитесь, что количество колонок и местоположения элементов сетки настраиваются так, как ожидается. Сетка должна иметь три колонки, если ширина экрана больше 900 пикселей, две – при ширине между 600 и 900 пикселями и одну, если ширина экрана меньше 600 пикселей (см. рис. 14.9).

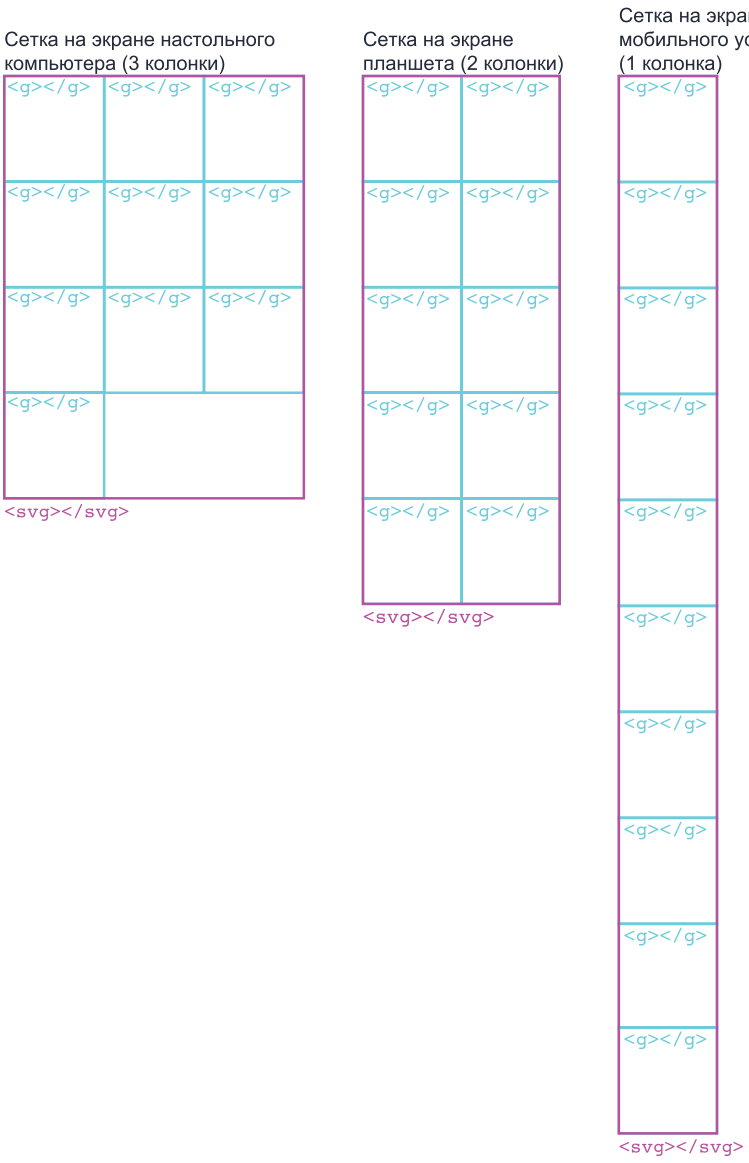


Рис. 14.9 Сетка SVG имеет три колонки на экранах с шириной больше 900 пикселей, две – при ширине от 600 до 900 пикселей и одну на экранах меньшего размера

14.5 Создание радиальных визуализаций

Подготовив основу проекта, можно начинать использовать D3 для создания визуализаций работ Ван Гога! В этом разделе мы построим множество небольших визуализаций, начав с осей и надписей, продолжив картинами и закончив рисунками и письмами.

14.5.1 Добавление радиальных осей

Основу наших небольших визуализаций можно упростить до круга и метки года. Но перед реализацией этих элементов нам нужно определить их точное местоположение. На рис. 14.10 показана схема, иллюстрирующая различные параметры, которые необходимо учитывать перед позиционированием круга и метки года. Мы уже рассчитали ширину (`smWidth`) и высоту (`smHeight`) каждой малой визуализации. Чтобы гарантировать наличие достаточного свободного места между визуализациями для меток месяцев, мы можем определить отступ вокруг каждого круга, скажем, равным 60 пикселям. Исходя из этого значения и ширины элементов сетки, можно вычислить радиус кругов.

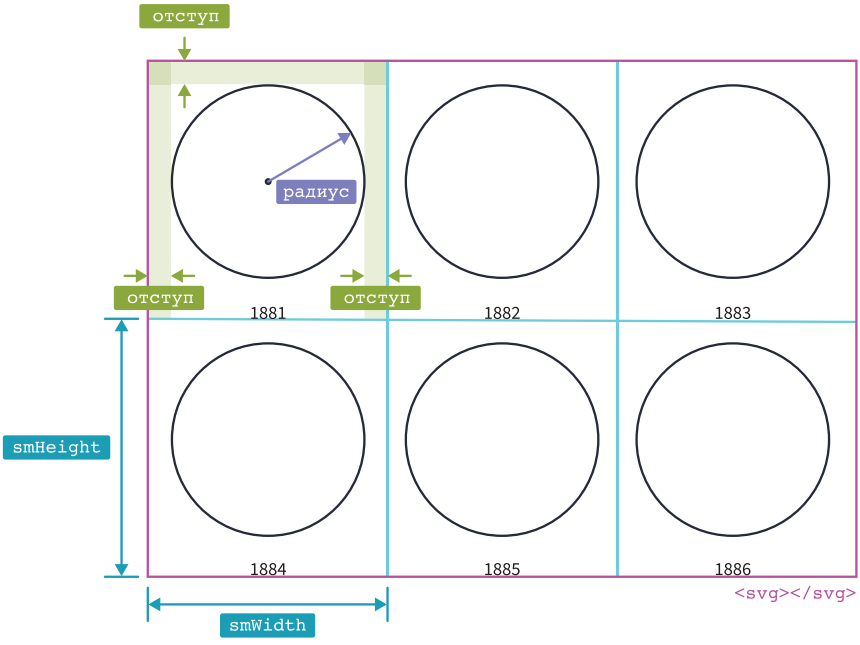


Рис. 14.10 Мы уже рассчитали ширину и высоту элементов сетки. Выбрав фиксированное значение отступа вокруг кругов, мы сможем вычислить их радиус

Начнем строить визуализацию в компоненте с именем `GridItem.svelte`, дочернем по отношению к компоненту `Grid.svelte`. В листинге 14.5 мы сначала импортируем этот компонент в `Grid.svelte`. Затем добавляем `GridItem` внутрь каждого блока, что приведет к созданию `GridItem` для каждого года из массива `years`. Мы передаем `smWidth`, `smHeight` и текущий год `year` в качестве свойств этому дочернему компоненту.

Листинг 14.5 Импорт компонента GridItem (Grid.svelte)

```
<script>
  import GridItem from "./GridItem.svelte";

  ...
</script>

{#if svgWidth && svgHeight}
  <svg width={svgWidth} height={svgHeight}>
    {#each years as year, i}
      <g transform="translate(
           {(i % numColumns) * smWidth},
           {Math.floor(i / numColumns) * smHeight})"
      >
        <rect x={0} y={0} width={smWidth} height={smHeight} />
        <GridItem {smWidth} {smHeight} {year} />
      </g>
    {/each}
  </svg>
{/if}
```

Импортировать компонент GridItem

Добавить компонент GridItem для каждого года внутрь соответствующей группы SVG. Передать smWidth, smHeight и текущий год как свойства

В листинге 14.6 мы начинаем работать внутри `GridItem.svelte` и импортируем свойства `smWidth`, `smHeight` и `year` в тегах `script`. Затем мы объявляем константу `padding` со значением `60` и вычисляем радиус круга на основе `padding` и `smWidth`. Поскольку радиус объявлен как реактивная переменная (`$`), он будет пересчитываться каждый раз при изменении `smWidth`.

В разметке мы используем два элемента группы, чтобы установить начало относительной системы координат для визуализаций. Первый из них смещается по горизонтали на половину `smWidth` и служит точкой отсчета для метки года, которую затем нужно перенести только по вертикали вниз. Второй элемент группы смещается по вертикали в центр круга. Эта стратегия пригодится нам, когда мы начнем добавлять в визуализации дополнительные фигуры для представления картин, рисунков и писем.

Листинг 14.6 Добавление круга и года (GridItem.svelte)

```
<script>
  export let smWidth;
  export let smHeight;
  export let year;

  const padding = 60;
  $: radius = (smWidth - 2 * padding) / 2;
</script>

<g transform="translate({smWidth / 2}, 0)">
  <g transform="translate(0, {padding + radius})">
    <circle cx={0} cy={0} r={radius} />
  </g>
  <text x={0} y={smHeight - 5} text-anchor="middle">{year}</text>
</g>

<style lang="scss">
  circle {
    fill: none;
    stroke: $text;
  }
</style>
```

Импортировать smWidth, smHeight и соответствующий год из родительского компонента

Установить отступ и вычислить радиус круга

Сместить группы SVG, чтобы они служили опорными точками для элементов, которые будут содержать

Следующий шаг – добавление осей и меток месяцев, как показано на рис. 14.11. Как видите, начало системы координат находится в центре круга благодаря смещениям групп SVG, выполненным ранее. Линии, обозначающие оси месяцев, будут начинаться в начале координат и заканчиваться на границе круга, причем для каждого месяца угол будет разным.

Чтобы вычислить местоположение конечных точек осей, нам придется вспомнить уроки тригонометрии. Возьмем в качестве примера ось февраля (Feb). Справа на рис. 14.11 можно видеть, что мы можем сформировать прямоугольный треугольник (треугольник, в котором один угол имеет угол 90°), соединив ось со смещениями точки по горизонтали (x) и вертикали (y). Давайте также обозначим символом «тета» (θ) угол между двенадцатью часами (0°) и осью февраля.

Из школьного курса тригонометрии мы помним, что синус угла θ равен x, деленному на длину оси февраля (т. е. радиусу круга). Соответственно, мы можем вычислить горизонтальную координату конечной точки, умножив радиус на $\sin\theta$. Аналогично косинус угла θ равен y, деленному на длину оси февраля. Соответственно, мы можем вычислить вертикальную координату конечной точки, умножив

радиус на cos θ и на -1, потому что мы движемся в отрицательном направлении вертикальной оси:

$$\sin\theta = x \div \text{радиус} \Rightarrow x = \text{радиус} \times \sin\theta,$$
$$\cos\theta = y \div \text{радиус} \Rightarrow y = \text{радиус} \times \cos\theta.$$

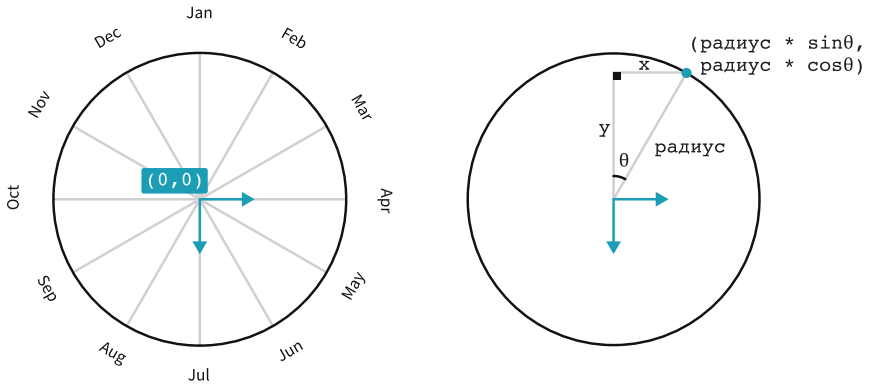


Рис. 14.11 Внутри круга нужно нарисовать ось для каждого месяца. Начальная точка этих осей совпадает с началом системы координат визуализации, местоположения конечных точек можно определить с помощью правил тригонометрии

Продолжим работать в `GridItem.svelte` и нарисуем 8 месяцев. В листинге 14.7 мы сначала объявляем точечную шкалу, которая принимает массив месяцев (доступен в файле `/utils/months.jsv`) и возвращает соответствующий угол. Нам нужно, чтобы январь (Jan) отображался на двенадцать часов, что соответствует углу 0°. Мы знаем, что полный круг – это 360°, или 2π радиан. Поскольку в году 12 месяцев, мы добавляем последний угол в нашу шкалу как 2π – 2π/12 радиан, или полный круг минус одну двенадцатую часть круга.

В разметке мы используем блок `each` для добавления одного элемента линии для каждого месяца. Начальная точка каждой линии – `(0, 0)`, а конечные точки формируются с помощью тригонометрических уравнений, представленных выше.

Листинг 14.7 Добавление осей месяцев (GridItem.svelte)

```
<script>
  import { scalePoint } from "d3-scale";
  import { months } from "../utils/months";

  export let smWidth;
  export let smHeight;
  export let year;
```

```
  const padding = 60;
  $: radius = (smWidth - 2 * padding) / 2;

  const monthScale = scalePoint()
    .domain(months)
    .range([0, 2 * Math.PI - (2 * Math.PI) / 12]);
</script>

<g transform="translate({smWidth / 2}, 0)">
  <g transform="translate(0, {padding + radius})">
    <circle cx={0} cy={0} r={radius} />
    {#each months as month}
      <line
        x1="0"
        y1="0"
        x2={radius * Math.sin(monthScale(month))}
        y2={-1 * radius * Math.cos(monthScale(month))}
        stroke-linecap="round"
      />
    {/each}
  </g>
<text x={0} y={smHeight - 5} text-anchor="middle">{year}</text>
</g>

<style lang="scss">
  circle {
    fill: none;
    stroke: $text;
  }
  line {
    stroke: $text;
    stroke-opacity: 0.2;
  }
</style>
```

Объявить точечную шкалу, которая принимает массив месяцев и возвращает соответствующий угол

Добавить элемент линии для каждого месяца. Начальная точка каждой линии – (0, 0), а конечная точка вычисляется с помощью тригонометрических уравнений

В качестве последнего шага добавим метку для каждой оси месяца, отступив 30 пикселей за пределы круга. В листинге 14.8 мы добавляем текстовый элемент для каждого месяца и с помощью JavaScript-метода `slice()` записываем первые три буквы из названия месяца. Чтобы правильно расположить текстовую метку, мы сначала перемещаем ее, а затем поворачиваем. Величину перемещения мы находим с помощью тригонометрических функций, используя те же формулы, что применялись для вычисления конечных точек осей. Для меток в верхней половине круга (между девятью и тремя часами) поворот производится на угол, соответствующий углу оси. Для меток в нижней половине (между тремя и девятью часами) добавляются дополнительные 180°, чтобы метки было легче читать.

Внутри функций JavaScript – `Math.sin()` и `Math.cos()` – мы используем радианы, но в атрибуте `transform`, описывающем поворот, ве-

личина угла должна указываться в градусах. Чтобы упростить перевод из радиан в градусы, мы создали вспомогательную функцию `radiansToDegrees()`, которую вы найдете в `/utils/helper.js`. Она принимает угол в радианах и возвращает тот же угол в градусах.

Листинг 14.8 Добавление меток месяцев (GridItem.svelte)

```
<script>
  import { scalePoint } from "d3-scale";
  import { months } from "../utils/months";
  import { radiansToDegrees } from "../utils/helpers";

  export let smWidth;
  export let smHeight;
  export let year;

  const padding = 60;
  $: radius = (smWidth - 2 * padding) / 2;

  const monthScale = scalePoint()
    .domain(months)
    .range([0, 2 * Math.PI - (2 * Math.PI) / 12]);
</script>

<g transform="translate({smWidth / 2}, 0)">
  <g transform="translate(0, {padding + radius})">
    <circle cx={0} cy={0} r={radius} />
    {#each months as month}
      <line
        x1="0"
        y1="0"
        x2={radius * Math.sin(monthScale(month))}
        y2={-1 * radius * Math.cos(monthScale(month))}
        stroke-linecap="round"
      />
      <text
       class="month-label"
       transform="translate(
        {(radius + 30) * Math.sin(monthScale(month))},
        {-1*(radius+30) * Math.cos(monthScale(month))}
       )
       rotate({
        monthScale(month) <= Math.PI / 2 ||
        monthScale(month) >= (3 * Math.PI) / 2
         ? radiansToDegrees(monthScale(month))
         : radiansToDegrees(monthScale(month)) - 180
       })"
       text-anchor="middle"
       dominant-baseline="middle">
         {month.slice(0, 3)}
       </text>
```

Переместить метку на 30 пикселей за пределы круга, используя те же тригонометрические формулы, что и для конечных точек осей

Добавить текстовый элемент для каждого месяца

Повернуть метки в верхней половине на тот же угол, что и соответствующие им оси. А при повороте меток в нижней части круга добавить 180° для удобства чтения

Сохранить только первые три буквы месяца

```
    {/each}
  </g>
<text x={0} y={smHeight - 5} text-anchor="middle">{year}</text>
</g>

<style lang="scss">
  circle {
    fill: none;
    stroke: $text;
  }
  line {
    stroke: $text;
    stroke-opacity: 0.2;
  }
  .month-label {
    font-size: 1.4rem;
  }
</style>
```

14.5.2 Применение силовой компоновки к элементам на окружности

После подготовки осей можно приступать к визуализации самих данных. Начнем с того, что представим каждую картину Ван Гога в виде круга. Круги будут сгруппированы по годам, а затем по месяцам создания, вокруг конечных точек соответствующих осей месяцев. Как мы договорились в разделе 14.3, цвет кругов будет определяться жанром картины, обводка кругов – техникой исполнения. Наконец, мы установим прямую зависимость между площадью кругов и размерами соответствующих им произведений.

На рис. 14.12 показано, чего нам нужно добиться. Чтобы сгенерировать кластеры кругов и при этом избежать их перекрытия, воспользуемся силовой компоновкой D3.

Прежде чем углубляться в код, давайте кратко обсудим архитектуру наших компонентов и выработаем стратегию наилучшего движения вперед. Наши небольшие визуализации состоят из трех слоев компонентов, как показано на рис. 14.13. Первый содержится в компоненте `Grid.svelte`, отвечающем за добавление контейнера SVG в разметку и размещение годов в виде сетки. Этот компонент «знает» обо всех годах, для которых мы создадим визуализацию.

Второй слой реализован в `GridItem.svelte`. Этот компонент знает только об одном годе, он отображает метку с номером года и оси месяцев. Наконец, в проекте есть компоненты `Paintings.svelte`, `Drawings.svelte` и `Letters.svelte`. Они находятся в папке `chart_components/`, но мы с ними пока не разбирались. Как следует из названий компонентов, они отвечают за визуализацию картин, рисунков и писем, созданных в течение года. Поскольку они будут вызываться из `GridItem.svelte`, они тоже будут знать только об одном годе данных.

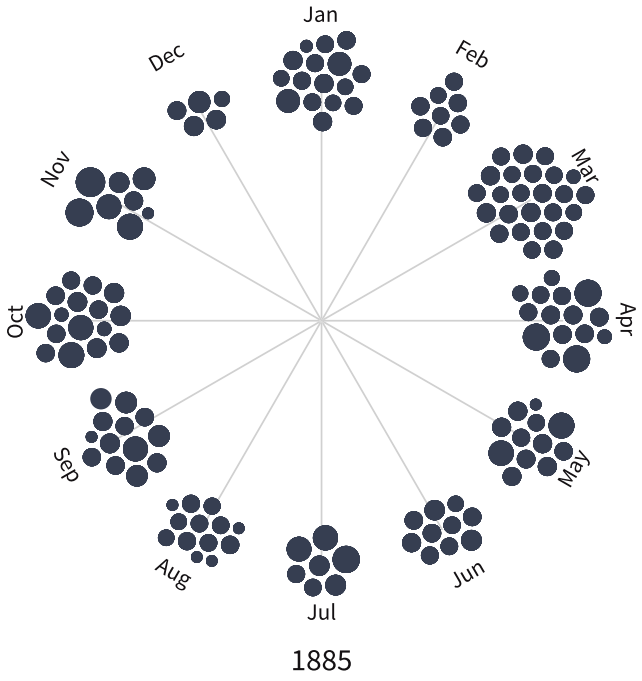


Рис. 14.12 Каждая картина, написанная Ван Гогом, будет представлена кругом. Мы разместим круги на небольших визуализациях, соответствующих годам их создания, и сгруппируем на концах осей месяцев. Для создания кластеров будет использоваться силовая компоновка D3

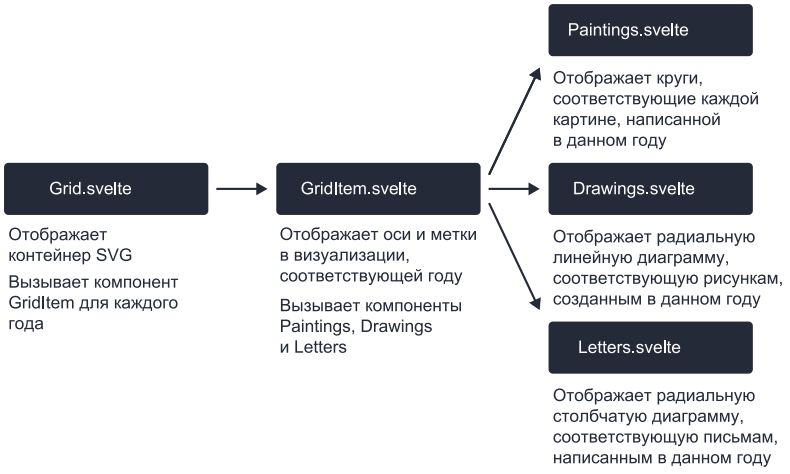


Рис. 14.13 Наши небольшие визуализации включают три слоя компонентов. Первый (`Grid.svelte`) отвечает за общую визуализацию и адаптивную сетку. Второй (`GridItem.svelte`) содержит каждую небольшую визуализацию, соответствующую году, и отображает оси месяцев. Последний (`Paintings.svelte`, `Drawings.svelte` и `Letters.svelte`) отвечает за элементы визуализации, связанные с картинами, рисунками и письмами

Учитывая такую многоуровневую организацию, лучшим местом для загрузки всего набора данных картин является компонент `Grid.svelte`, потому что он контролирует всю визуализацию и загружается только один раз. Затем этот компонент передаст соответствующие картины в виде входных свойств в `GridItem.svelte`, который затем передаст их в `Paintings.svelte`.

Основываясь на этой логике, мы импортируем в `Grid.svelte` набор данных с картинами (листинг 14.9). Поскольку позже нам понадобится задать размеры кругов пропорционально размерам картин, мы вычислим площади этих работ и используем эту информацию для поиска самых больших в наборе данных. Обратите внимание, что в наборе данных есть несколько картин, размеры некоторых неизвестны. Их мы будем представлять кругами с радиусом, равным 3 пикселям.

Чтобы преобразовать площади картин (в $см^2$) в площади кругов на экране (в px^2), можно использовать линейную шкалу. Назовем эту шкалу `paintingAreaScale` и для ее определения найдем максимальную площадь, охватываемую диапазоном, с помощью формулы площади круга:

$$a = \pi r^2.$$

Наконец, мы передадим в `GridItem` данные и функции, необходимые для отображения картин. Обратите внимание, как фильтруется набор данных, чтобы извлечь из него только картины, соответствующие текущему году.

Листинг 14.9 Создание шкалы для преобразования площадей картин в площади кругов (Grid.svelte)

```
<script>
  import { range, max } from "d3-array";
  import { scaleLinear } from "d3-scale";
  import paintings from "../data/paintings.json";

  ...

  paintings.forEach((painting) => {
    if (painting.width_cm && painting.height_cm) {
      painting["area_cm2"] = painting.width_cm *
        painting.height_cm;
    }
  });
  const maxPaintingArea = max(paintings, (d) =>
    d.area_cm2);

  const maxPaintingRadius = 8;
  const paintingDefaultRadius = 3;
  const paintingAreaScale = scaleLinear()
    .domain([0, maxPaintingArea])
    .range([0,Math.PI*Math.pow(maxPaintingRadius,2)]);
```

Рассчитать площадь каждой картины и найти максимальную площадь картины в наборе данных

Объявить линейную шкалу, принимающую площадь картины и возвращающую соответствующую площадь круга SVG для отображения на экране

```
</script>

<svelte:window bind:innerWidth={windowWidth} />

{#if svgWidth && svgHeight}
  <svg width={svgWidth} height={svgHeight}>
    {#each years as year, i}
      <g
        transform="translate(
           {(i % numColumns) * smWidth},
           {Math.floor(i / numColumns) * smHeight})"
      >
        <rect x={0} y={0} width={smWidth} height={smHeight} />
        <GridItem
          {smWidth}
          {smHeight}
          {year}
          {paintingAreaScale}
          {paintingDefaultRadius}
          paintings={paintings.filter((painting) =>
            painting.year === year)}
        />
      </g>
    {/each}
  </svg>
{/if}
```

Передать данные и функции, которые понадобятся для отображения картин, в GridItem. Отфильтровать набор данных, оставив только картины, соответствующие текущему году

В `GridItem.svelte` нам нужно объявить свойства, полученные из `Grid.svelte`, импортировать компонент `Paintings`, добавить компонент `Paintings` в разметку и передать ему те же свойства, как показано в листинге 14.10.

Листинг 14.10 Вызов компонента Paintings (GridItem.svelte)

```
<script>
  import Paintings from "../chart_components/
    Paintings.svelte";

  export let paintingAreaScale;
  export let paintingDefaultRadius;
  export let paintings;

  ...
</script>

<g transform="translate({smWidth / 2}, 0)">
  <g transform="translate(0, {padding + radius})">
    <circle ... />
    {#each months as month}
       <line ... />
       <text ... >{month.slice(0, 3)}</text>
    {/each}
```

Импортировать компонент Paintings

Объявить новые свойства, полученные от Grid.svelte

```
      <Paintings
        {paintingAreaScale}
        {paintingDefaultRadius}
        {paintings}
        {monthScale}
        {radius}
      />
    </g>
    <text ... >{year}</text>
</g>
```

Вызвать компонент Paintings и передать ему требуемые свойства

Главное действие происходит в `Paintings.svelte`. Здесь выполняется обход картин, полученных во входных свойствах, и для каждой из них в разметку добавляется круг. Начальное местоположение кругов – конец соответствующей оси месяца, который можно найти с помощью тригонометрических уравнений, использованных ранее. Мы также должны учесть тот факт, что месяц создания некоторых картин неизвестен. Такие картины мы поместим в центр визуализации.

Чтобы вычислить радиус кругов, мы вызываем `paintingAreaScale` (см. листинг 14.11). Эта шкала возвращает площадь, поэтому нам нужно вычислить радиус по следующей формуле:

$$r = \sqrt{a \div \pi}.$$

Листинг 14.11 Добавление кругов, представляющих картины (Paintings.svelte)

```
<script>
  export let paintingAreaScale;
  export let paintingDefaultRadius;
  export let paintings;
  export let monthScale;
  export let radius;
</script>
```

Объявить свойства, полученные от GridItem.svelte

```
{#each paintings as painting}
  <circle
    cx={painting.month !== ""
     ? radius * Math.sin(monthScale(painting.month))
     : 0}
    cy={painting.month !== ""
     ? -1 * radius * Math.cos(monthScale(
        painting.month))
     : 0}
    r={painting.area_cm2
     ? Math.sqrt(paintingAreaScale(painting.area_cm2)
         / Math.PI)
     : paintingDefaultRadius}
  />
{/each}
```

Обойти картины и для каждой добавить круг

Расположить круги на конце соответствующей оси месяца, используя тригонометрические формулы, обсуждавшиеся ранее. Если месяц создания картины неизвестен, то поместить ее в центр визуализации

Вызвать paintingAreaScale, чтобы получить площадь круга, соответствующую размерам картины, и вычислить радиус круга. Если размеры картины недоступны, то установить радиус круга равным значению по умолчанию

На данный момент круги картин перекрываются, как показано на рис. 14.14. Мы исправим этот недостаток далее, используя силовую компоновку D3.

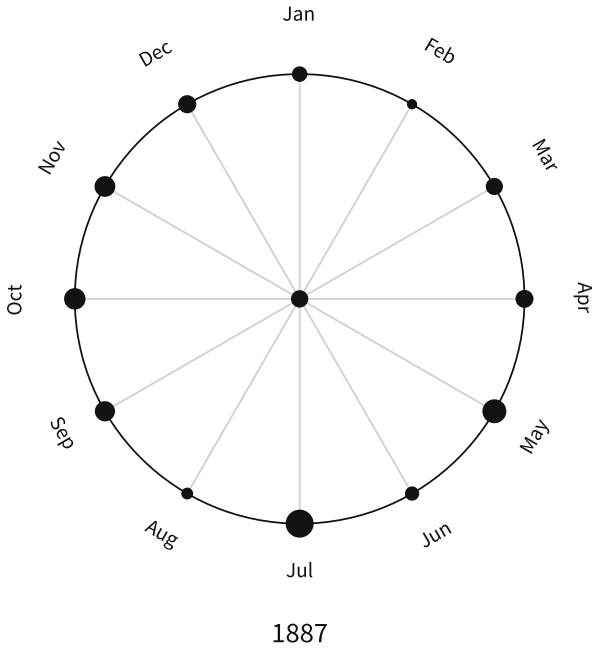


Рис. 14.14 На данный момент круги картин перекрываются. Мы исправим этот недостаток, используя силовую компоновку D3

Чтобы создать кластеры узлов на концах осей месяцев, мы используем силовую компоновку D3. Эта компоновка довольно сложна, поэтому всем, кто хотел бы получить более глубокое введение, мы рекомендуем прочитать главу 12. Код в листинге 14.12 инициализирует новую симуляцию системы сил вызовом метода `forceSimulation()`, которому передается массив картин `paintings`. Здесь также объявляется пустой массив узлов `nodes`, которые будут обновляться после каждой итерации. Далее код выполняет цикл по этому массиву `nodes` и добавляет круги в разметку.

Силы, воздействующие на узлы, вычисляются внутри реактивного блока (`$`), вследствие чего перерасчет будет автоматически запускаться при изменении связанных переменных. Внутри блока силы позиционирования (`forceX` и `forceY`) толкают узлы к концам соответствующих осей месяцев, а сила столкновения (`forceCollide`) гарантирует отсутствие перекрытия между узлами.

В этой реализации мы также уменьшили `alpha` («температуру» симуляции) и увеличили скорость остывания `alphaDecay`, чтобы помочь симуляции сойтись быстрее. Эти значения были подобраны

методом проб и ошибок. Наконец, мы используем свойства x и y, добавленные в узлы симуляцией, чтобы задать атрибуты cx и cy соответствующих кругов.

Листинг 14.12 Получение позиций кругов (Paintings.svelte)

```
<script>
  import { forceSimulation, forceX, forceY, forceCollide } from "d3-force";

  ...

  let simulation = forceSimulation(paintings);
  let nodes = [];
```

Инициализировать новую симуляцию и пустой массив nodes

```
  simulation.on("tick", () => {
    nodes = simulation.nodes();
  });
```

После каждой итерации обновить массив nodes последними вычисленными позициями

```
  $: {
    simulation
```

Использовать реактивный блок для запуска перерасчета сил при изменении связанной переменной

```
      .force("x",
        forceX((d) => d.month !== ""
          ? radius * Math.sin(monthScale(d.month))
          : 0
        ).strength(0.5)
      )
      .force("y",
        forceY((d) => d.month !== ""
          ? -1 * radius * Math.cos(monthScale(d.month))
          : 0
        ).strength(0.5)
      )
```

Применить силы позиционирования, притягивающие круги к концам осей месяцев. Если месяц не определен, то использовать центр визуализации

```
      .force("collide",
        forceCollide()
          .radius((d) => d.width_cm === null
          && d.height_cm === null
            ? paintingDefaultRadius + 1
            : Math.sqrt(paintingAreaScale(d.area_cm2)
              / Math.PI) + 1
        ).strength(1)
      )
```

Применить силу столкновения, чтобы избежать перекрытия узлов

```
      .alpha(0.5)
      .alphaDecay(0.1);
  }
</script>
```

Уменьшить параметр «температуры» alpha и увеличить скорость «остывания», чтобы помочь симуляции сойтись быстрее

```
{#each nodes as node}
  <circle
    cx={node.x}
    cy={node.y}
    r={node.area_cm2
```

Использовать свойства x и y, вычисленные симуляцией для каждого узла, чтобы задать атрибуты cx и cy кругов

```
        ? Math.sqrt(paintingAreaScale(node.area_cm2) / Math.PI)
        : paintingDefaultRadius}
  />
{/each}
```

Теперь вы должны увидеть сформировавшиеся кластеры узлов на концах осей месяцев. Чтобы завершить визуализацию картин, установим цвет кругов, исходя из жанров картин. Файл `utils/subjects.js` содержит массив доступных жанров и их цветов. В листинге 14.13 мы объявляем порядковую шкалу, которая принимает жанр и возвращает соответствующий цвет. Нам остается только установить атрибут `fill` кругов, вызвав эту шкалу.

Листинг 14.13 Установка цвета кругов (Paintings.svelte)

```
<script>
  import { scaleOrdinal } from "d3-scale";
  import { subjects } from "../utils/subjects";

  ...

  const colorScale = scaleOrdinal()
    .domain(subjects.map((d) => d.subject))
    .range(subjects.map((d) => d.color));
</script>

{#each nodes as node}
  <circle
    cx={node.x}
    cy={node.y}
    r={node.area_cm2
       ? Math.sqrt(paintingAreaScale(node.area_cm2) / Math.PI)
       : paintingDefaultRadius}
    fill={colorScale(node.subject)}
  />
{/each}
```

Объявить порядковую шкалу, которая принимает жанр и возвращает соответствующий цвет

Установить атрибут `fill` кругов, вызвав цветовую шкалу

Мы закончили визуализацию картин! В настоящий момент визуализация должна выглядеть, как показано на рис. 14.15.

14.5.3 Рисование радиальной диаграммы с областями

Наш следующий шаг – нарисовать радиальную диаграмму с областями, представляющую количество рисунков, выполненных Ван Гогом в течение каждого года. В главе 4 мы узнали, как использовать генератор форм D3 для вычисления атрибута `d` элементов `path` для линейных диаграмм и диаграмм с областями. Здесь мы воспользуемся похожей стратегией и генератором форм `lineRadial()` из модуля d3-shape.

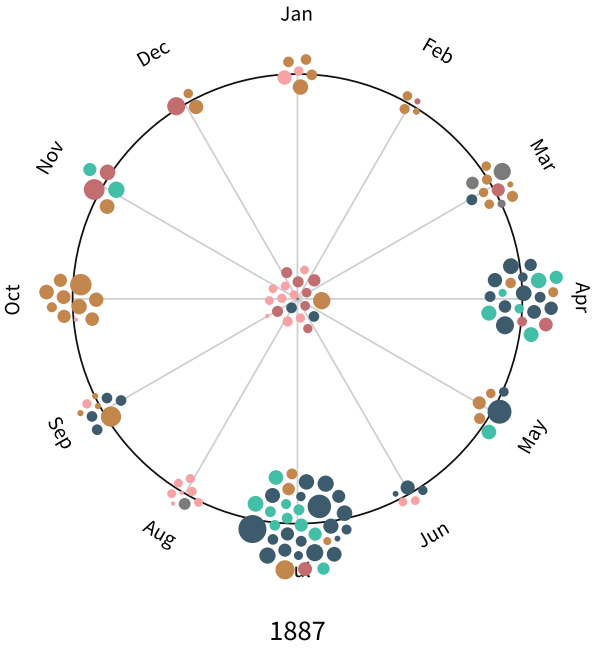


Рис. 14.15 Визуализация картин, созданных Ван Гогом в 1887 году

Как и в предыдущем разделе, мы учтем наличие трех слоев компонентов Svelte в нашей визуализации. Весь набор данных с рисунками мы загрузим в компоненте `Grid.svelte` и там же вычислим максимальное количество рисунков, нарисованных за месяц. Затем разделим информацию по годам, как показано в листинге 14.14, и передадим получившийся набор данных в компонент `GridItem.svelte`. Внутри `GridItem.svelte` мы инициализируем шкалу, отвечающую за вычисление местоположений на осях месяцев, соответствующих количеству рисунков (листинг 14.15), и все это передадим в компонент `Drawings.svelte`, который нарисует диаграмму с областями.

Листинг 14.14 Импорт набора данных с информацией о рисунках и его реорганизация (Grid.svelte)

```
<script>
  import drawings from "../data/drawings.json";
  import { months } from "../utils/months";

  ...

  const yearlyDrawings = [];
  years.forEach((year) => {
    const relatedDrawings = { year: year, months: [] };
    months.forEach((month) => {
      relatedDrawings.months.push({
        month: month,
```

Реорганизовать набор данных с информацией о рисунках, чтобы сгруппировать их по годам и месяцам

```
        drawings: drawings.filter(drawing =>
          drawing.year === year.toString() &&
          drawing.month === month),
      });
    });
    yearlyDrawings.push(relatedDrawings);
  });

  const maxDrawings = max(yearlyDrawings, d =>
    max(d.months, (i) => i.drawings.length)
  );
</script>

<svelte:window bind:innerWidth={windowWidth} />

{#if svgWidth && svgHeight}
  <svg width={svgWidth} height={svgHeight}>
    {#each years as year, i}
      <g
        transform="translate(
           {(i % numColumns) * smWidth},
           {Math.floor(i / numColumns) * smHeight})"
      >
        <rect x={0} y={0} width={smWidth} height={smHeight} />
        <GridItem
          {smWidth}
          {smHeight}
          {year}
          {paintingAreaScale}
          {paintingDefaultRadius}
          paintings={paintings.filter((painting) =>
            painting.year === year)}
          {maxDrawings}
          drawings={yearlyDrawings.find((d) =>
            d.year === year).months}
        />
      </g>
    {/each}
  </svg>
{/if}
```

Реорганизовать набор данных с информацией о рисунках, чтобы сгруппировать их по годам и месяцам

Найти максимальное количество рисунков, нарисованных за месяц

Передать maxDrawings и рисунки текущего года в GridItem

Код в листинге 14.15 инициализирует шкалу, отвечающую за вычисление позиции на оси месяца, соответствующей количеству рисунков.

Листинг 14.15 Шкала для диаграммы с областями, представляющей количество рисунков (GridItem.svelte)

```
<script>
  import { scaleLinear } from "d3-scale";
  import Drawings from "../chart_components/Drawings.svelte";
```

```
  export let maxDrawings;
  export let drawings;

  ...

 $: radialScale = scaleLinear()
    .domain([0, maxDrawings])
    .range([0, 2 * radius]);
</script>

<g transform="translate({smWidth / 2}, 0)">
  <g transform="translate(0, {padding + radius})">
    <circle ... />
    ...
    <Drawings {drawings} {monthScale} {radialScale} />
  </g>
  <text ... >{year}</text>
</g>
```

Инициализировать линейную шкалу, которая принимает количество рисунков и возвращает соответствующую позицию на оси месяца

Добавить компоненты Drawings и передать им необходимые свойства

В листинге 14.16 мы используем метод `lineRadial()` из библиотеки D3, чтобы инициализировать генератор линий. Как объяснялось в главе 4, его функции доступа настраиваются для вычисления координат каждой точки данных. Но на этот раз мы работаем в полярных координатах, а не в декартовых, отсюда необходимость в использовании функций `angle()` и `radius()`. Добавляя элемент `path` в разметку, мы вызываем генератор линий и с его помощью устанавливаем атрибут `d`. В стилях для созданной кривой задается полупрозрачность заливки. На рис. 14.16 показана диаграмма с областями для 1885 года.

Листинг 14.16 Рисование радиальных диаграмм с областями (Drawings.svelte)

```
<script>
  import { lineRadial, curveCatmullRomClosed } from "d3-shape";

  export let drawings;
  export let monthScale;
  export let radialScale;

  const lineGenerator = lineRadial()
    .angle((d) => monthScale(d.month))
    .radius((d) => radialScale(d.drawings.length))
    .curve(curveCatmullRomClosed);
</script>

<path d={lineGenerator(drawings)} />

<style lang="scss">
  path {
    fill: rgba($secondary, 0.25);
    pointer-events: none;
  }
</style>
```

Использовать метод lineRadial() для инициализации генератора линий. Настроить функции доступа angle() и radius() для вычисления полярных координат каждой точки данных

Вызвать генератор линий, чтобы задать атрибут d элемента path, используемого для рисования областей

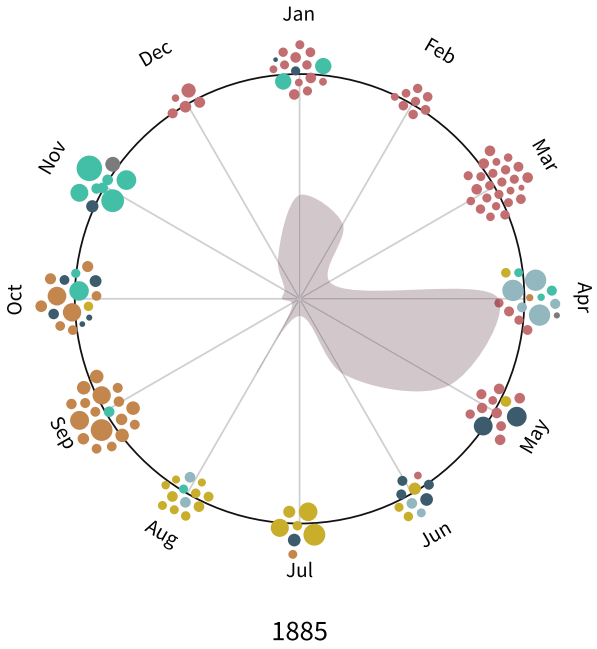


Рис. 14.16 Для построения радиальной диаграммы с областями, представляющей количество рисунков, созданных Ван Гогом в течение каждого года, используется генератор линий

14.5.4 Рисование радиальной столбчатой диаграммы

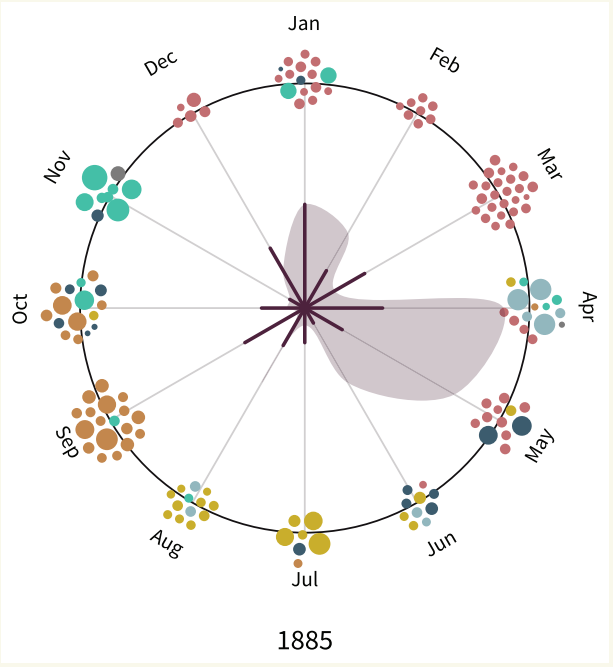
Последний набор данных, который мы должны визуализировать, – это количество писем, написанных Ван Гогом в разные месяцы. Поскольку теперь у вас есть все необходимые знания, попробуйте сделать это самостоятельно!

Упражнение: визуализация количества писем, написанных Ван Гогом в разные месяцы, в виде радиальной столбчатой диаграммы

Мы предлагаем вам самим выполнить эту часть проекта. Вы можете сделать это совершенно самостоятельно или последовать нашим инструкциям:

1. загрузите набор данных с письмами в `Grid.svelte`. Этот набор данных содержит общее количество писем, написанных в каждом месяце;
2. передайте письма, соответствующие текущему году, в `GridItem.svelte` через входные свойства;
3. в `GridItem.svelte` импортируйте компонент `Letters`. Добавьте его в разметку и передайте ему шкалы и данные о письмах в виде входных свойств;

4 в `Letters.svelte` добавьте для каждого месяца линию, начинающуюся в начале координат и заканчивающуюся в точке, соответствующей количеству писем. Для вычисления координат конечной точки используйте тригонометрические функции.



Количество писем, написанных Ван Гогом в течение 1885 года

Если у вас возникнут трудности или вы захотите сравнить свое решение с нашим, то вы найдете его в разделе D.10 и в папке `14.5.4-Radial_bar_chart/end` в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно. Ваше решение может немного отличаться от нашего, и это нормально!

Завершая статическую версию визуализации, закомментируем прямоугольники и круги, которые использовались ранее, чтобы показать сетку макета, и добавим временную шкалу. Поскольку временная шкала не имеет отношения к D3, мы не будем объяснять ее реализацию, но желающие найдут соответствующий код в листинге D.14.4 и в папке `14.5.4` в репозитории этой главы. Вы также можете воспринять это как вызов и построить ее самостоятельно! На рис. 14.17 показана окончательная статическая версия визуализации с временной шкалой.

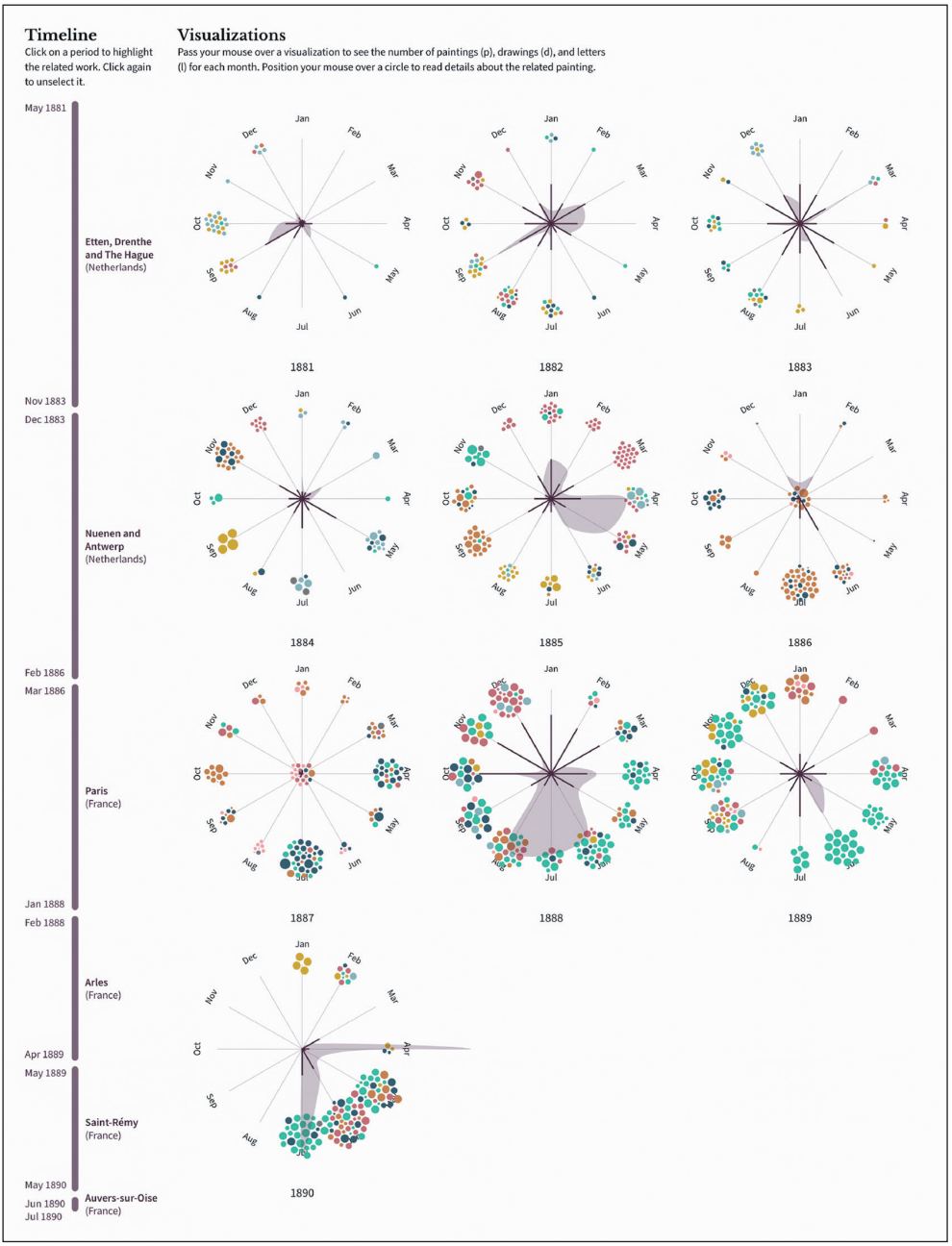


Рис. 14.17 Окончательная статическая версия визуализации с временной шкалой, показывающей, где жил Ван Гог в период с 1881 по 1890 год, а также визуализации его художественного творчества по годам

14.6 Выбор значимых взаимодействий

Теперь, закончив статическую часть проекта, отступим на шаг назад и подумаем, как будущие пользователи могли бы захотеть его исследовать. Какую дополнительную информацию они будут искать? Какие вопросы задавать? Можем ли мы ответить на них с помощью взаимодействий? Вот, например, три вопроса, которые может задать пользователь:

- Какая картина представлена каждым кругом? Можно ли ее увидеть?
- Как узнать, сколько рисунков и писем было нарисовано и написано в июне 1885 года? Примерное количество можно оценить по легенде, но хотелось бы видеть конкретные цифры.
- Как связать города, в которых жил Ван Гог, показанные на временной шкале, и его художественное творчество?

Мы можем ответить на эти вопросы, реализовав простые взаимодействия: на первые два – с помощью всплывающих подсказок, а на последний – с помощью перекрестного выделения. Поскольку эта глава и без того получилась довольно длинной, а сами взаимодействия мало связаны с D3 (при использовании сторонних фреймворков мы стараемся не использовать обработчики событий D3, чтобы исключить воздействие D3 на дерево DOM), мы не будем вдаваться в детали реализации. Основная цель данного раздела – показать, как планировать взаимодействия, которые имеют смысл для проекта. Вы можете поиграть с этими взаимодействиями в онлайн-версии проекта (http://mng.bz/z8rg), а исходный код вы найдете в папке `14.6` в репозитории данной главы. Также работа этих взаимодействий показана на рис. 14.18, 14.19 и 14.20.



Рис. 14.18 Подсказка, появляющаяся при наведении указателя мыши на круг, представляющий картину, и раскрывающая детали этой картины

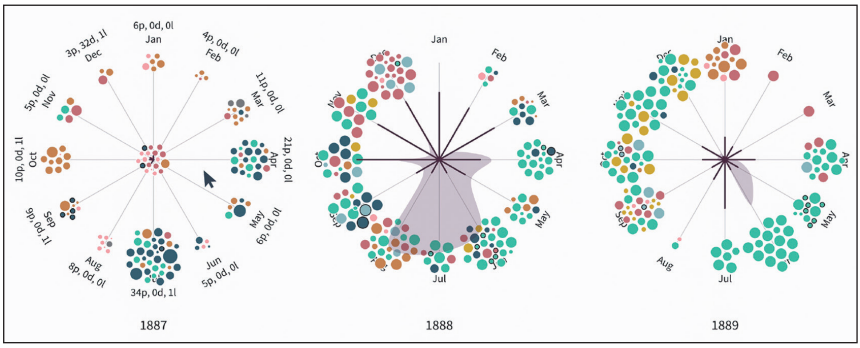


Рис. 14.19 Подсказка, появляющаяся при наведении указателя мыши на визуализацию и показывающая количество картин, рисунков и писем за каждый месяц

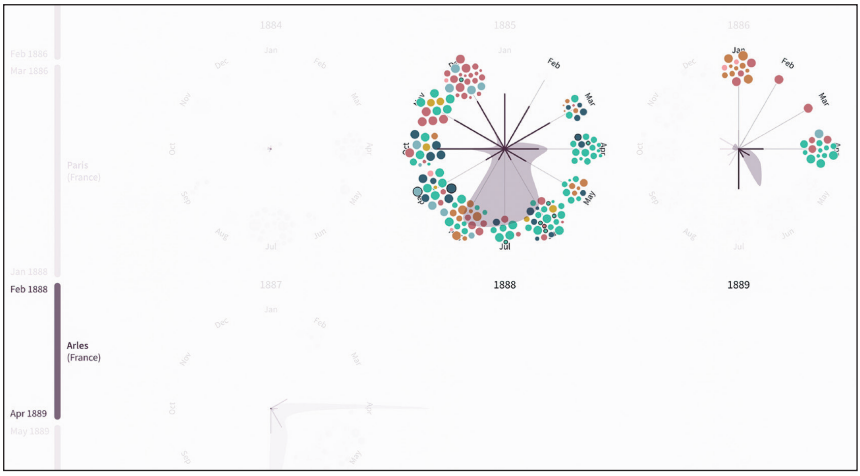


Рис. 14.20 При выборе периода на временной шкале отображаются только картины, созданные в этот период

На этом мы завершаем работу над проектом! Надеемся, он вдохновил вас на творческий подход к визуализации. Если вы хотите глубже изучить использование D3 и Svelte для создания интерактивных визуализаций данных, мы настоятельно рекомендуем подписаться на курс «Better Data Visualizations with Svelte» Коннора Ротшильда (Connor Rothschild; http://mng.bz/jXVV).

Интервью с Зейной Вольф

Вольф – ученый, учитель, специалист по визуализации данных и скалолаз.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

Я довольно разносторонний человек. В свое время изучала биологию и прикладную физику в бакалавриате, имела опыт исследований в области брачного поведения веслоногих рачков, занималась сравнительными функциями филогенетических алгоритмов и астрофизикой нейтронных звезд, а мои кандидатские проекты представляли собой франкенштейновскую мешанину из биомеханики, мягкой робототехники и гидродинамики. Я не была уверена в том, что именно хотела бы исследовать, поэтому использовала любые возможности и пробовала себя в разных областях.

Обучаясь в аспирантуре, я поняла, что больше всего мне нравится заниматься исследованиями, погружающими меня в изучение новой темы или инструмента, заставляющими что-то создавать, изучать данные и стараться донести суть до других. Так что в основном мне нравится заниматься исследованиями ради исследований, без привязки к какой-то конкретной теме. С одной стороны, это приносит настоящее наслаждение, но с другой – это не очень хорошо для карьеры в академической среде, где нужно выбрать конкретную тему исследования.

Летом 2020 года я принимала участие в виртуальной онлайн-конференции для преподавателей-ассистентов за несколько дней до начала семестра. На одном из семинаров нам всем дали по одной минуте, чтобы описать курсы, которые мы собираемся преподавать. Когда один из преподавателей-ассистентов описал свой курс, я сразу же написала ему в чате, попросив сообщить более подробную информацию. Это был курс «CS 171: введение в визуализацию данных». Я подумала, что курс с таким названием выглядит не только полезным, но и интересным, а кроме того, он соответствовал моему последнему требованию к курсу для получения кандидатской степени, поэтому я записалась на него в тот же день.

Я нашла в визуализации данных все, что мне так нравится в науке, – постоянное погружение в новые темы для изучения данных, создание визуализаций с нуля, исследование новых инструментов и навыков и сообщение открытий другим. Я сразу же почувствовала себя в этой области как дома и быстро влюбилась в работы Надии Бремер и Ширли Ву, которые демонстрировались на протяжении всего курса.

По окончании курса я начала работать в учебной лаборатории в Центре преподавания и обучения имени Дерека Бока в Гарварде в качестве стипендиата по медиа и дизайну. Здесь я близко познакомилась с дизайном и разработкой веб-сайтов. Интерактивные онлайн-визуализации данных должны размещаться на каком-то сайте, поэтому мне

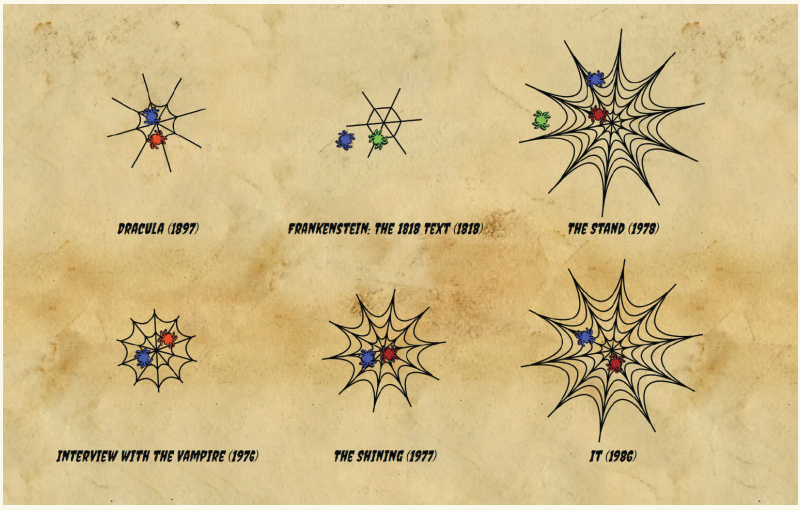
показалось это стоящим вложением своего времени. Я провела год, работая с инженерным отделом над созданием визуализаций данных для их факультета, с профессором над созданием веб-сайта для его курса и с учебной лабораторией, где я проводила собственные семинары, обучая студентов основам визуализации данных, от гештальт-принципов восприятия до применения конкретных инструментов, таких как Mapbox. После этого я в основном работала над личными проектами, где мне была интересна тема или требовалось освоить новый навык либо инструмент.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашем рабочем процессе визуализации данных?

Мой рабочий процесс создания визуализаций во многом зависит от конечного продукта. Если я создаю простые диаграммы для исследований или для академической публикации, или занимаюсь каким-либо статистическим моделированием, то использую R или Python. Если мне нужно быстро создать прототип диаграммы или панель мониторинга с простыми диаграммами, то использую Tableau. При создании сложной или интерактивной визуализации я обращаюсь к D3. И наконец, создавая неинтерактивные визуализации неизменяющихся исходных данных, я могу использовать Affinity Designer.

Выбор наиболее подходящего инструмента может избавить от множества неприятностей. Даже если вы решили специализироваться на определенном инструменте, например D3, то простое знание альтернатив, их плюсов и минусов может оказаться неплохим подспорьем.

Представленный проект: «Horror Novels» (www.zanewolf.com/projects/horror-novels)



В этом проекте Зейн Вульф визуализирует 20 лучших романов ужасов по версии Goodreads

Опишите, пожалуйста, ваш проект «Horror Novels».

«Horror Novels» – это дань моему любимому празднику, Хэллоуину, и одной из моих главных вдохновительниц в изучении визуализации данных и D3, Ширли Ву, и ее визуализации «Film Flowers» (https://shirleywu.studio/filmflowers/). Я взяла 20 лучших романов ужасов по версии Goodreads и визуализировала каждую отдельную книгу в виде паутины. Характеристики паутины, такие как размер, количество радиальных и концентрических нитей и даже количество пауков в паутине, определяются данными из каждой книги, такими как количество страниц, жанры, возраст и общая оценка книги по пятибалльной шкале.

Расскажите, как создавался этот проект.

Первоначально я хотела визуализировать данные по бестселлерам New York Times, но, к сожалению, найденные мной наборы данных либо охватывали только последнее десятилетие, либо в них отсутствовала информация, которую я надеялась найти, например жанр, рейтинг, количество страниц и т. д. В большинстве случаев идеального набора данных не существует, и вам придется собирать его из нескольких источников. Я использовала набор данных Zenodo Best Books Ever с данными из огромного списка Best Books Ever, содержащего более 50 000 книг. Затем я нашла список Goodread Definitive Horror Book List и использовала его, чтобы сократить набор данных с 50 000 книг до 20.

Получив исходные данные, я сделала несколько своих набросков паутины и поискала изображения паутины в Google, чтобы получить представление о разнообразии дизайна. Какие аспекты паутины я могла бы сопоставить с данными, что можно было бы изменить с точки зрения дизайна и в конечном итоге получить паутину? Я остановилась на размере, количестве радиальных и концентрических нитей и «обвислости» паутины (т. е. степени натянутости концентрических колец, соединяющих радиальные нити). Наконец, я решила сопоставить жанр с цветом, но паутина не отличается своей яркостью. Я держала в уме несколько вариантов, но только когда закончила реализацию паутины, я решила использовать разноцветных пауков. На этапе создания набросков не нужно зацикливаться на мелких деталях визуализации. Я исследовала данные в R и выяснила, как преобразовать исходные данные, чтобы использовать их для создания веб-сайтов (например, преобразовать дату публикации в возраст).

Следующей частью процесса было веб-программирование. Я разбила процесс создания паутины на три этапа. Сначала я представила подготовленные данные (например, возраст, жанр, рейтинг, число страниц) в виде переменных для построения диаграмм (таких как радиус, количество радиальных нитей, угол между радиальными нитями, количество концентрических нитей), используя различные шкалы D3 (линейную, квантильную, пороговую). Выбор конкретной шкалы зависел от распределения значений данных. Я выбрала те, которые создавали наибольшее визуальное разнообразие. Рейтинги 20 лучших романов

ужасов находятся в диапазоне от 3,8 до 4,3 по пятибалльной шкале, поэтому мне пришлось крепко подумать, как выжать максимум разнообразия из диапазона оценок 0,5. Кроме того, по мере создания паутины я меняла соответствие между элементами данных и аспектами паутины, чтобы добиться визуального эффекта, который я хотела получить. Не бойтесь возвращаться к чертежной доске и пересматривать план игры!

Далее мне нужно было написать функции для создания кривых SVG, из которых образуется паутина. Я много экспериментировала, чтобы понять, как перейти от переменных, таких как радиус и угол, к настоящим паутинам. И да, мне пришлось гуглить базовые тригонометрические формулы. Используя координаты центров, сгенерированные функциями, которые вычисляли местоположение каждой паутины на странице, а также радиус и угол между радиальными нитями, я смогла создать элементы `path` для радиальных нитей. Затем, используя радиус, угол, координаты центров, количество концентрических нитей и степень обвислости, я смогла вычислить, где должны располагаться концентрические нити (на каком расстоянии от центра) и насколько провисшими они должны быть. И еще одну функцию я использовала для вычисления координат пауков, чтобы максимизировать разнообразие, ограничивая при этом перекрытие. Я не стала создавать своих пауков, а нашла в интернете симпатичную SVG-иконку с изображением паука. Собирая автомобиль, не бойтесь покупать шины на шинном заводе, вместо того чтобы изготавливать их самостоятельно.

Поскольку визуализация задумывалась статической и не должна была учитывать возможное обновление исходных данных, этап рисования паутин и пауков был довольно прост по сравнению с предыдущим шагом: я лишь добавила в разметку несколько элементов `path` и пауков. Создание легенды с помощью D3 может быть одновременно и сложным, и монотонным делом, потому что нужно было создать данные для легенды и нарисовать представление каждого значения каждой переменной. Но как только контент появился на странице, я занялась его оформлением: добавила фон в виде старого пергамента и шрифт, как на старых афишах фильмов-ужасов категории B. Этот шаг превратил визуализацию из коллекции паутин в то, что я ожидала увидеть на плакате в библиотеке.

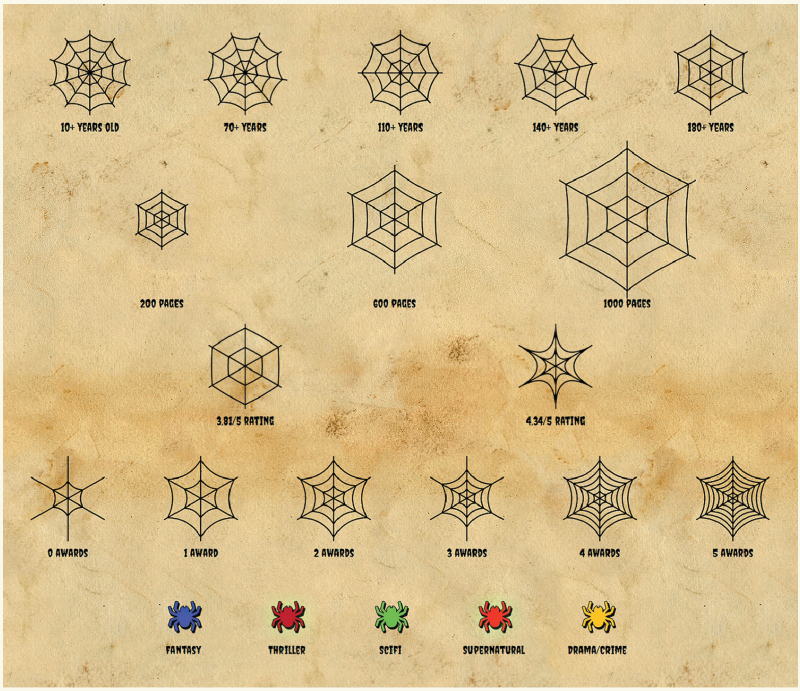


Рис. 14.21 Визуализация данных о романах ужасов с использованием паутин разного размера, с разным количеством радиальных и концентрических нитей и разной степени «обвислости», а также цвета пауков

Помимо всего прочего, я проверила все используемые цвета на доступность для людей с ограниченными возможностями! Конечно, мне не удалось создать полностью доступную цветовую палитру (я была ограничена набором цветов, связанных с Хэллоуином), но вообще всегда желательно занимать стратегическую позицию и назначать проблемные цвета переменным, наименее представленным в наборе данных.

Последний этап процесса был самым коротким, но он принес мне массу удовольствия. Я вернулась к коду и добавила несколько интересных случайностей и пасхальных яиц. Например, я добавила поворот паутины и пауков на случайный угол, чтобы при каждом обновлении страницы ориентации каждой паутины и каждого паука менялись. Эти маленькие штрихи делают визуализацию уникальной для каждого человека, который ее видит, даже притом что визуализация не является интерактивной или обновляемой.

На своем сайте вы рассказываете, что использование псевдокода помогло вам спланировать рисование паутин на экране. Расскажите подробнее об этом подходе и его преимуществах.

Псевдокод помогает описать алгоритм на естественном языке. Каждый язык программирования имеет свой синтаксис. Вы можете забыть точный

способ записи оператора `if` или цикла `for` в языке R, Python или MATLAB, но без труда опишете цель на естественном языке: «для каждого элемента массива сделать то-то и то-то». Запись алгоритмов на псевдокоде перед программированием помогает сосредоточиться на том, чего вы хотите добиться. Это особенно полезно в ситуациях, когда еще не знаете, какие шаги нужно выполнить, чтобы перейти от входных данных к желаемому результату. Например, мне нужна была функция, которая принимала бы несколько входных значений, таких как радиус и угол, и возвращала значение атрибута `d` для элемента `path`, изображающего паутину. Конечно же, в первый момент я не знала, как это сделать. Но разработка псевдокода помогла мне разобраться с математикой, а затем понять, как реализовать математические вычисления в виде программного кода, который, получая входные значения радиуса и угла, вычислял необходимые кривые.

Иногда псевдокод записывается прямо в редакторе как последовательность комментариев к отдельным шагам. Если это простая или короткая функция, я так и поступаю. После этого я перехожу к началу, пишу код под каждым комментарием и в результате получаю прекрасно откомментированную реализацию! Но иногда нужно разобраться со сложными математическими вычислениями или что-то нарисовать. В таких случаях я беру ручку и пишу псевдокод на бумаге.

На последнем этапе этого процесса я переношу псевдокод в реальный код с соответствующим синтаксисом и пунктуацией. У меня такой склад ума, что я запоминаю лишь несколько базовых функций и основной синтаксис любого языка, поэтому я частенько гуглю способы реализации тех или иных действий. Но, имея псевдокод, я уже понимаю, чего хочу, а записанные в псевдокоде шаги – это мои готовые поисковые запросы. В результате весь процесс оказывается очень эффективным.

Совет сначала понять, что вы собираетесь выразить в программном коде, может показаться очевидным, но, как показывает практика, слишком легко сесть за компьютер и почувствовать себя идиотом, который не знает, как писать код, потому что не знает, какие шаги нужно выполнить, чтобы прийти к желаемому результату. Это одна из ситуаций, когда «проще предотвратить, чем исправить», особенно если вы еще только учитесь писать код и чувствуете, что программирование дается вам с трудом. Слишком легко разочароваться в программировании, и описание алгоритмов на псевдокоде – это секретное оружие, которое поможет избежать негатива с самого начала. Псевдокод помогает отделить задачу поиска решения от задачи реализации этого решения на языке программирования. Просто начиная писать код, не имея плана, вы пытаетесь решить обе задачи одновременно. Для меня лично это очень сложно, и создание псевдокода – моя палочка-выручалочка, значительно облегчающая процесс разработки.

Какая из книг, представленных в визуализации, нравится вам больше всего? Моя, без сомнений, «Оно» Стивена Кинга!

Весьма иронично, но я не фанатка ужасов! Единственная из этих книг, которую я читала, – «Франкенштейн», и это было так давно, что я помню

только общий сюжет. Полагаю, что ее и можно было бы назвать любимой. Когда я росла, у меня было достаточно времени, чтобы читать ради удовольствия, и дополнительные тревоги – это было последнее, что я хотела испытывать. Повзрослев, я чувствую то же самое. Но за многие годы я просмотрела некоторые фильмы, и особенно мне запомнился «Молчание ягнят».

Над какими проектами вы надеетесь работать в будущем?

В этом плане у меня нет никакой уверенности. Я все еще чувствую себя новичком в этой области и продолжаю знакомиться с огромным разнообразием инструментов визуализации данных и их возможностями. Я надеюсь найти такую работу, где смогу заняться самыми разными проектами, выяснить границы своих интересов и отыскать более конкретный ответ на этот вопрос. Впрочем, меня увлекает наука, и мне хотелось бы заняться наведением мостов между академическими и практическими сферами.

Итоги

- Одно из главных преимуществ D3 – возможность создавать инновационные визуализации.
- Процесс создания визуализации обычно включает следующие шаги: сбор данных, подготовка и исследование данных, создание наброска макета визуализации, построение основы проекта, реализация элементов визуализации и добавление взаимодействий.
- Имеющиеся навыки программирования на JavaScript можно с успехом использовать для извлечения данных из веб-страниц.
- При изучении данных полезно перечислить имеющиеся в них количественные и качественные атрибуты, потому что для их визуализации используются разные приемы.
- Для создания инновационных визуализаций иногда нужно разбить желаемые приемы на строительные блоки, а для этого часто полезно знать, какие модули D3 содержат методы, которые могут пригодиться для реализации этих блоков.
- При создании радиальных визуализаций вычисления производятся в полярной системе координат с привлечением базовых тригонометрических функций.
- Чтобы спланировать содержательное взаимодействие, спросите себя, на какие вопросы пользователь захочет получить ответы, изучая вашу визуализацию.

15 Отображение визуализаций с помощью Canvas

В этой главе рассматриваются:

- плюсы и минусы использования Canvas вместо SVG;
- отображение простых фигур с помощью Canvas;
- применение смешанного режима отображения для визуализации данных;
- изучение стратегии управления взаимодействиями с Canvas.

В этой книге мы визуализировали данные с помощью элементов SVG. Благодаря присущей им возможности обеспечить четкую визуализацию на экранах всех размеров и разрешений их легко сделать адаптивными, доступными, производительными и интерактивными. Но в некоторых ситуациях, например когда в DOM добавляется слишком много элементов SVG (более 1000), производительность SVG может резко упасть, что делает HTML-элемент Canvas лучшим выбором.

В этой главе мы сравним SVG с Canvas и выясним, когда лучше использовать тот или другой механизм. Затем мы увидим, как реализовать отображение простых геометрических фигур в элементе Canvas, повторив упражнение «Галерея фигур SVG» из главы 1. Мы также рассмотрим возможность смешанного режима отображения для улучшения производительности проекта из главы 14, отобразив

круги картины с помощью Canvas вместо SVG. Наконец, мы рассмотрим приемы реализации взаимодействий с визуализацией на основе Canvas.

15.1 Что такое Canvas и когда его использовать

SVG и Canvas – это две технологии рисования в веб-браузерах. Но, в отличие от технологии SVG, генерирующей векторную графику и использующей *декларативный* подход (мы сообщаем браузеру желаемый результат, а не как его получить), Canvas генерирует растровую (пиксельную) графику и использует более *императивный* подход (мы сообщаем браузеру, как рисовать, вызывая команды JavaScript).

Для визуализации цифровых данных обычно предпочтительнее использовать SVG, потому что большинство диаграмм не подразумевает добавления большого количества элементов в DOM. Но при работе с большими наборами данных и большим количеством визуализаций или сложными анимациями на одной странице имеет смысл обратить внимание на Canvas. Как правило, если нужно добавить в DOM более тысячи элементов, то, возможно, для создания визуализации лучше использовать Canvas. Но помните, что это ограничение продолжает расширяться по мере увеличения производительности браузеров и компьютеров.

В табл. 15.1 перечислены основные факторы, которые следует учитывать, выбирая между SVG и Canvas. Основные причины, по которым выбор падает на SVG, – четкий вид генерируемых изображений и доступность каждого отдельного элемента в DOM, что позволяет оформлять графику с использованием стилей CSS, добавлять в нее анимацию и взаимодействовать с ней. С другой стороны, Canvas дает преимущества при работе со сложной графикой, включающей цветовые градиенты и многочисленные анимации, и может значительно повысить производительность.

Таблица 15.1 Сравнение SVG и Canvas

Свойство	SVG	Canvas
Качество изображения	(+) Векторные изображения: четко выглядят на экранах любых размеров, разрешений и уровней масштабирования	(–) Растровые изображения: имеют тенденцию выглядеть немного размытыми, особенно при увеличении масштаба или на экранах с высоким разрешением. Эта проблема особенно ярко наблюдается, когда визуализация включает текстовые метки
Производительность	(+) Невысокая производительность при создании большого количества элементов, но достаточно высокая производительность в других случаях	(+) Очень высокая производительность отображения сложной анимированной графики. При создании небольших и простых визуализаций графика SVG, как правило, имеет лучшую производительность

Таблица 15.1 (окончание)

Свойство	SVG	Canvas
Поддержка стилей CSS	(+) Да. SVG поддерживает переходы и анимацию CSS	(–) Нет
Возможность взаимодействий	(+) К элементам SVG можно подключать обработчики событий	(–) Обработчики событий можно подключить к самому элементу Canvas, но не к рисункам внутри него. Для добавления интерактивности приходится прибегать к дополнительным стратегиям
Анимации	(+) SVG поддерживает возможность создания анимационных эффектов с помощью CSS и JavaScript, но не поддерживает сложные анимации, например такие как в играх	(+) Подходит для обработки большого количества анимаций со сложными деталями и градиентами
Отладка	(+) Каждый элемент SVG доступен в DOM, им можно управлять и исследовать	(–) В DOM присутствует только сам элемент Canvas, и с помощью инспектора браузера нельзя управлять сгенерированным изображением или исследовать его
Доступность	(+) SVG упрощает создание доступного контента, поскольку все элементы SVG доступны в DOM, а текстовые элементы SVG доступны программам чтения с экрана	(–) Элементу Canvas можно присвоить атрибут `aria-label`, но в остальном он обрабатывается программами чтения с экрана как обычное растровое изображение

Вот несколько проектов визуализации данных, которые используют исключительно Canvas или смешанный подход SVG-Canvas. Используйте инспектор в вашем браузере, чтобы увидеть, где создатели применяли Canvas, а не SVG:

- «An Interactive Visualization of Every Line in Hamilton», Ширли Ву (Shirley Wu; https://pudding.cool/2017/03/hamilton/);
- «Mapping Diversity», студия Sheldon (https://mappingdiversity.eu/greece/athens/);
- «The Inside Scoop of Ben & Jerry's», Хешам Эйсса (Hesham Eissa) и Линдси Поултер (Lindsey Poulter) (https://benjerry.heshlinds-dataviz.com/).

15.2 Отображение простых фигур с помощью Canvas

Прежде чем перейти к сложной графике, посмотрим сначала, как рисовать простые фигуры. Для этого вернемся к упражнению «Галерея фигур SVG» из главы 1 и повторим его с помощью Canvas. Готовый проект вы найдете по адресу http://mng.bz/WEqd. Знакомясь с концепцией контекста Canvas и основными командами, мы нарисуем линию, прямоугольники, круг, эллипс и кривую (см. рис. 15.1).

Чтобы приступить к выполнению этого упражнения, откройте Visual Studio Code (VS Code) и перейдите в папку `start`, загружен-

ную с http://mng.bz/8wGW. Запустите локальную среду разработки, щелкните на надписи «Go Live» в строке состояния. Если эта надпись отсутствует, то, возможно, вы еще не установили расширение Live Server. Подробные инструкции по его установке вы найдете в приложении А.

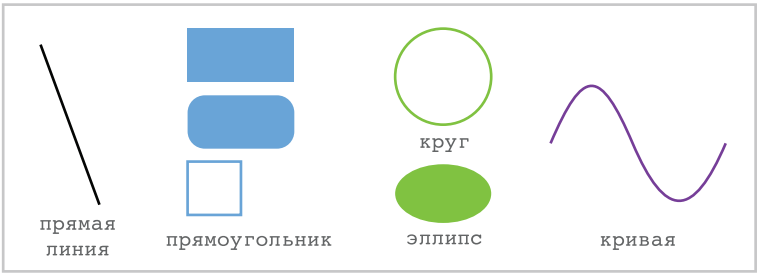


Рис. 15.1 Используя основные команды, можно рисовать линии, прямоугольники, круги и кривые

15.2.1 Элемент <canvas>

HTML-элемент `<canvas>` используется в сочетании с JavaScript для рисования на экране двухмерной графики. Также для рисования двух- и трехмерной графики с аппаратным ускорением можно использовать Web Graphics Library (WebGL) API, но этот способ мы не будем рассматривать.

Файлы для этого упражнения включают HTML-файл (`index.html`), содержащий разметку SVG, созданную в главе 1. В конце этого файла также имеется `div` с идентификатором `"canvas"`. Именно сюда мы внедрим наш элемент `canvas`. В папке `/js` вы найдете файл с именем `main.js`. В него мы поместим наш код. Обратите внимание, что библиотека D3 и `main.js` уже загружаются в `index.html` через тег `<script>`.

Прежде чем начать рисовать, добавим элемент `canvas` в DOM. В листинге 15.1 мы используем D3 для выбора `div` с идентификатором `"canvas"`, добавляем в него элемент `canvas` и добавляем для него черную рамку толщиной 1 пиксель.

Листинг 15.1 Добавление элемента canvas в DOM (main.js)

```
const canvas = d3.select("#canvas")
  .append("canvas")
    .style("border", "1px solid black");
```

Если открыть проект в браузере и заглянуть в инструмент инспектора, то можно увидеть, что элемент `canvas` имеет размер по умолчанию около 300×150 пикселей (см. рис. 15.2).

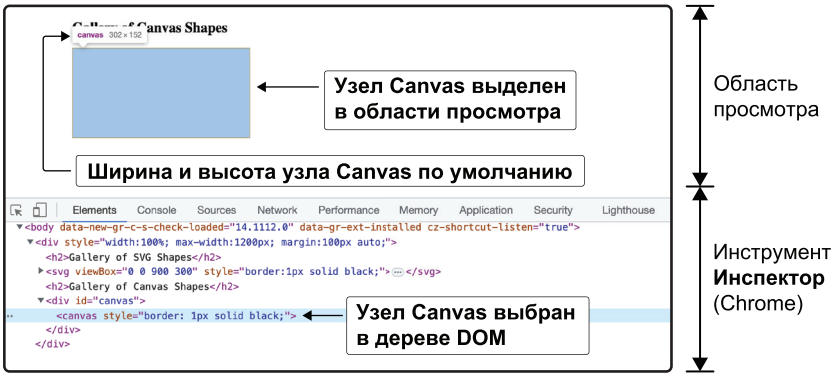


Рис. 15.2 По умолчанию элемент canvas имеет размер около 300×150 пикселей

Чтобы задать ширину и высоту, можно было бы использовать CSS, но поскольку Canvas отображает растровую графику, это может привести к размытию изображения. Поэтому лучше установить размер элемента `canvas` через его атрибуты `width` и `height`.

Код в листинге 15.2 расширяет листинг 15.1. Он выбирает `div` с идентификатором `"canvas"` и сохраняет его в константу с именем `container`. Затем задает атрибуты `width` и `height` элемента `canvas`. Ширина равна ширине контейнера, а высота составляет одну треть от ширины.

Листинг 15.2 Установка атрибутов width и height (main.js)

```
const canvas = d3.select("#canvas")
  .append("canvas")
    .style("border", "1px solid black");

const container = document.querySelector("#canvas");

const width = container.offsetWidth;
const height = 0.333 * width;

canvas
  .attr("width", width)
  .attr("height", height);
```

Выбрать div с идентификатором "canvas" и сохранить в константе с именем container

Установить атрибуты `width` и `height` элемента canvas. Ширина равна ширине родительского div, а высота – одной трети ширины

Вернувшись в браузер и изменив размер экрана, вы увидите, что элемент canvas теперь адаптируется к размеру родительского элемента, как показано на рис. 15.3.

Наконец, нам нужно рассмотреть еще один важный аспект – плотность пикселей. Некоторые экраны имеют более высокую плотность пикселей, чем другие, т. е. они вмещают больше пикселей на квадратный сантиметр. Вероятно, вы уже знаете, что растровые изображения выглядят размытыми на таких экранах, если только не удвоить

или не утроить их разрешение. Нам нужно принять эту стратегию для создания графики Canvas.

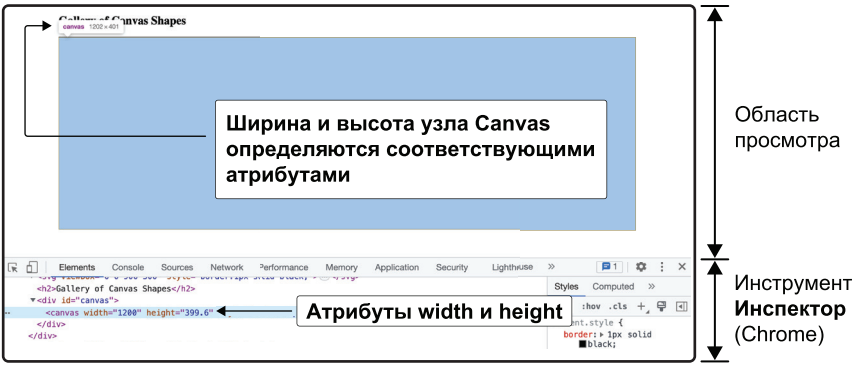


Рис. 15.3 Атрибуты `width` и `height` определяют размеры элемента `canvas`

К счастью, сделать это очень просто! Объект `window` имеет свойство `devicePixelRatio`, которое содержит отношение разрешения в *логических пикселях* к разрешению в *физических пикселях* текущего устройства. Это свойство возвращает единицу для традиционных экранов и двойку для экранов с высоким разрешением (High Dots Per Inch, HiDPI). В листинге 15.3 мы просто умножаем ширину и высоту на это значение, а затем ограничиваем CSS-свойство `max-width` элемента `canvas` величиной 100 %. При работе с экраном Retina свойство `devicePixelRatio` имеет значение 2, поэтому мы удваиваем ширину и высоту элемента `canvas`. Но поскольку он не может быть шире своего родителя, плотность логических пикселей внутри элемента `canvas` окажется вдвое больше, и разрешение графики будет скорректировано.

Листинг 15.3 Исправление разрешения (main.js)

```
const canvas = d3.select("#canvas")
  .append("canvas")
    .style("border", "1px solid black")
    .style("max-width", "100%");

const container = document.querySelector("#canvas");
const devicePixelRatio = window.devicePixelRatio;

const setCanvasSize = () => {
  const width = container.offsetWidth;
  const height = 0.333 * width;

  canvas
    .attr("width", width * devicePixelRatio)
    .attr("height", height * devicePixelRatio);
```

Гарантировать, что элемент `canvas` не станет шире своего родителя

Получить свойство `devicePixelRatio` объекта `window` и сохранить его в константе

Умножить атрибуты `width` и `height` элемента `canvas` на отношение логических пикселей к физическим

```
};

setCanvasSize();
window.addEventListener("resize", setCanvasSize);
```

И последнее, что нужно сделать перед началом рисования, – инициализировать контекст элемента `canvas`. Контекст – это объект со свойствами и методами, которые мы используем далее для рисования. В листинге 15.4 мы берем узел, связанный с выборкой D3, и устанавливаем его контекст рисования равным `"2d"`. Если бы мы решили использовать WebGL, то присвоили бы значение `"webgl"`. Полученный контекст мы сохраняем в константу `context`, чтобы иметь простую возможность вызывать его.

Поскольку мы корректируем количество логических пикселей, отображаемых в элементе Canvas, необходимо применить масштабирующее преобразование к единицам Canvas с помощью метода `scale()`. По умолчанию одна единица Canvas соответствует 1 пикселю. Если применить преобразование `scale(2, 2)`, то каждая единица Canvas будет соответствовать 2 пикселям, а размеры фигур будут вдвое превышать их нормальный размер.

Листинг 15.4 Настройка контекста Canvas (main.js)

```
const canvas = d3.select("#canvas")
  .append("canvas")
    .style("border", "1px solid black")
    .style("max-width", "100%");
...

const context = canvas.node().getContext("2d");
context.scale(devicePixelRatio, devicePixelRatio);
```

Получить узел, выбранный с помощью D3, и настроить его контекст рисования, присвоив значение "2d"

Применить горизонтальное и вертикальное масштабирования к единицам Canvas

Теперь можно начинать рисовать! Рисование в Canvas производится с помощью команд JavaScript, применяемых к контексту, созданному в листинге 15.4. К счастью, имена этих команд говорят сами за себя.

15.2.2 Линия

Линия в Canvas – это аналог элемента `path` в SVG, поэтому мы вызываем команду `beginPath()`, чтобы начать новую линию. Если бы ранее мы нарисовали другие линии, то эта команда также сбросила бы контекст предыдущих команд рисования линий. В листинге 15.5 мы устанавливаем начальную точку линии командой `moveTo()`, которая принимает координаты `x` и `y`. Это как взять ручку, поднять ее

и поставить на лист бумаги в точке с координатами (66, 60). Затем, выполняя команду `lineTo()`, мы ведем ручку по листу бумаги, чтобы нарисовать линию до точки с координатами (186, 300). Наконец, командой `context.stroke()` очерчиваем проведенную линию. Обратите внимание, что здесь используются те же координаты начальной и конечной точек, что и при рисовании линии SVG, включенной в файл HTML. Шаги рисования линии с помощью Canvas показаны на рис. 15.4.

Листинг 15.5 Рисование линии (main.js)

```
context.beginPath();
context.moveTo(66, 60);
context.lineTo(186, 300);
context.stroke();
```

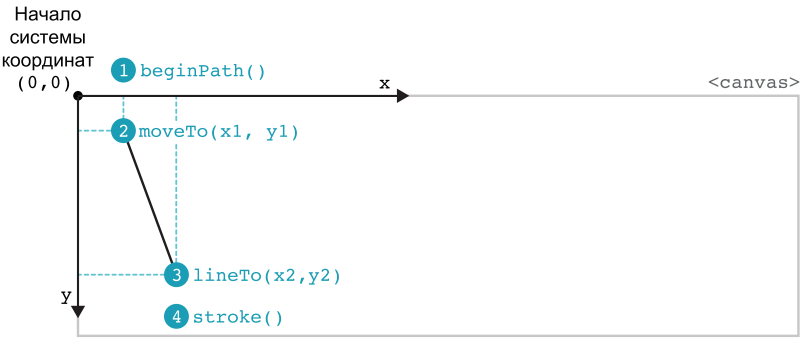


Рис. 15.4 Чтобы нарисовать линию с помощью Canvas, нужно выполнить четыре команды: `beginPath()`, чтобы задать начало линии, `moveTo()`, чтобы опустить перо в начальную точку, `lineTo()`, чтобы задать конец линии, и `stroke()`, чтобы фактически нарисовать линию

15.2.3 Прямоугольник

Рисование прямоугольника выглядит еще проще. Его форма определяется одной командой `rect()`, которая принимает четыре аргумента:

- `x` – горизонтальная координата верхнего левого угла прямоугольника;
- `y` – вертикальная координата верхнего левого угла прямоугольника;
- `width` – ширина прямоугольника;
- `height` – высота прямоугольника.

Следующий фрагмент рисует первый прямоугольник в нашей галерее фигур. Верхний левый угол этого прямоугольника находится в точке (346, 33), его ширина составляет 160 пикселей, а высота –

80 пикселей. Обратите внимание, что на этот раз для фактического рисования использована команда `fill()` вместо `stroke()`. Как нетрудно догадаться, команда `fill()` производит заливку фигуры, а команда `stroke()` рисует границы:

```
context.rect(346, 33, 160, 80);
context.fill();
```

Сохранив код и открыв проект в браузере, вы увидите черный прямоугольник, как на рис. 15.5. Как и в SVG, в Canvas по умолчанию используется черный цвет. Чтобы использовать другой цвет, нужно задать свойство `fillStyle` контекста перед рисованием фигуры, как показано ниже:

```
context.rect(346, 33, 160, 80);
context.fillStyle = "#6ba5d7";
context.fill();
```

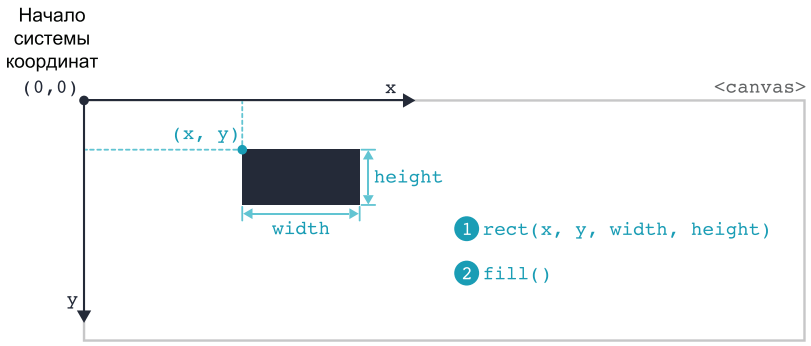


Рис. 15.5 Чтобы нарисовать прямоугольник, мы используем команды `rect()` и `fill()`. Мы также могли бы использовать `stroke()`, чтобы добавить обводку

Нарисовать прямоугольник со скругленными углами можно командой `roundRect()`. Она принимает те же аргументы, что и `rect()`, а также массив радиусов закруглений углов. Если в массиве передать только одно значение, то все четыре угла будут иметь одинаковый радиус закругления.

Обратите внимание, как в листинге 15.6 мы объединили рисование первых двух прямоугольников. Сначала мы объявляем их размеры, задаем свойство `fillStyle` контекста и после этого рисуем сразу два прямоугольника командой `fill()`. Затем мы объявляем третий прямоугольник, для которого рисуем только границу. Для этого мы задаем свойство `strokeStyle` контекста и вызываем команду `stroke()`.

Обратите также внимание, что мы использовали команду `beginPath()`, чтобы сбросить текущий контекст рисования линии. В Canvas рамки рисуются как линии. Если не сбросить предыдущий кон-

текст рисования линии (который мы настроили в листинге 15.5), то новая рамка автоматически унаследует стили из текущего контекста `strokeStyle`.

Листинг 15.6 Рисование прямоугольников (main.js)

```
context.rect(346, 33, 160, 80);
context.roundRect(346, 133, 160, 80, [20]);
context.fillStyle = "#6ba5d7";
context.fill();

context.beginPath();

context.rect(346, 233, 106, 80);
context.strokeStyle = "#6ba5d7";
context.stroke();
```

Объявить два прямоугольника – второй с закругленными углами

Настроить свойство fillStyle() контекста, задав синий цвет, и нарисовать первые два прямоугольника

Сбросить контекст рисования линии

Объявить новый прямоугольник, настроить свойство strokeStyle() контекста, задав синий цвет, и нарисовать рамку прямоугольника

Как видите, фигуры с похожими стилями можно группировать и рисовать их все сразу. Каждый раз, когда нужно поменять цвет, мы настраиваем свойства `fillStyle` и/или `strokeStyle` контекста. На данный момент элемент Canvas должен содержать три прямоугольника, как на рис. 15.6.

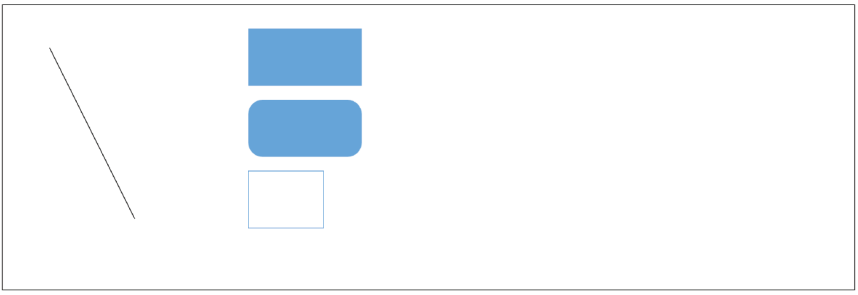


Рис. 15.6 Мы использовали ряд команд, чтобы нарисовать линию и три прямоугольника

15.2.4 Круг

В Canvas нет метода, рисующего круги, но есть метод для рисования дуг. Метод `arc()` принимает пять аргументов:

- `x` – горизонтальная координата центра дуги;
- `y` – вертикальная координата центра дуги;
- `radius` – радиус дуги;
- `startAngle` – угол начальной точки дуги (в радианах);
- `endAngle` – угол конечной точки дуги (в радианах).

Если начальный угол совпадает с конечным углом, то получается круг! Эллипсы рисуются с помощью похожей команды: `ellipse(x, y, radiusX, radiusY, startAngle, endAngle)`, где `radiusX` и `radiusY` –

радиусы в горизонтальном и вертикальном направлениях, как показано на рис. 15.7.

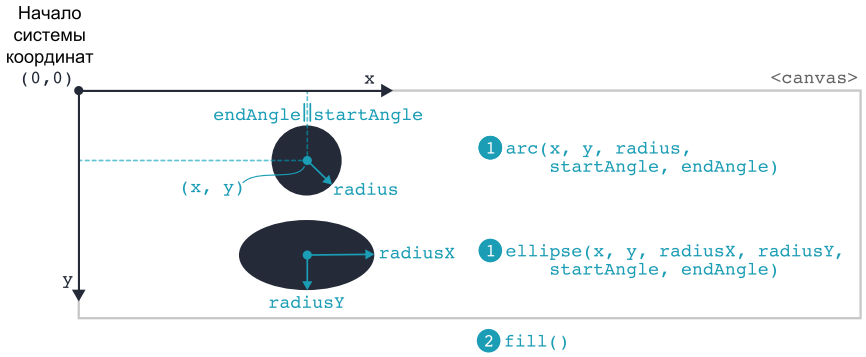


Рис. 15.7 Круги рисуются с помощью команды `arc()`, а эллипсы – с помощью `ellipse()`

В листинге 15.7 мы рисуем обводку круга и заполненный эллипс. Чтобы получить замкнутые формы, мы передаем в `startAngle` значение 0, а в `endAngle` – значение 2π радиан, что эквивалентно 360°. Мы также настраиваем свойство `lineWidth`, чтобы задать ширину обводки.

Обратите внимание, что перед рисованием и круга, и эллипса нам вновь пришлось с помощью `beginPath()` сбросить текущий контекст рисования линий. Попробуйте закомментировать эти строки, чтобы увидеть, что произойдет!

Листинг 15.7 Рисование круга и эллипса (main.js)

```
context.beginPath();
context.arc(706, 106, 66, 0, 2 * Math.PI);
context.strokeStyle = "#81c21c";
context.lineWidth = 3;
context.stroke();
```

Сбросить текущий контекст рисования линий и нарисовать обводку круга

```
context.beginPath();
context.ellipse(706, 273, 66, 40, 0, 0, 2 * Math.PI);
context.fillStyle = "#81c21c";
context.fill();
```

Сбросить текущий контекст рисования линий и нарисовать заполненный эллипс

15.2.5 Кривая

Для рисования сложных контуров в SVG мы часто используем генератор форм D3. В Canvas можно напрямую использовать атрибут `d`, возвращаемый этими генераторами. Canvas API предоставляет конструктор `Path2D()`, который принимает атрибут `d` контура SVG. Если

объект, созданный этим конструктором, передать свойству `stroke()` контекста, как показано в листинге 15.8, то будет нарисована та же фигура, которую мы получили бы в SVG.

Листинг 15.8 Рисование кривой (main.js)

```
const path = new Path2D("M900 200 C 945 110, 965 110,
➥ 1010 200 S 1075 293, 1120 200");
context.strokeStyle = "#773b9a";
context.stroke(path);
```

Создать новый объект Path2D, передав конструктору значение атрибута d соответствующего SVG-элемента path

Нарисовать кривую, передав полученный объект свойству stroke() контекста

На данный момент наша галерея фигур должна содержать линию, прямоугольники, круг, эллипс и кривую, как показано на рис. 15.8. В следующем подразделе мы добавим текстовые метки.

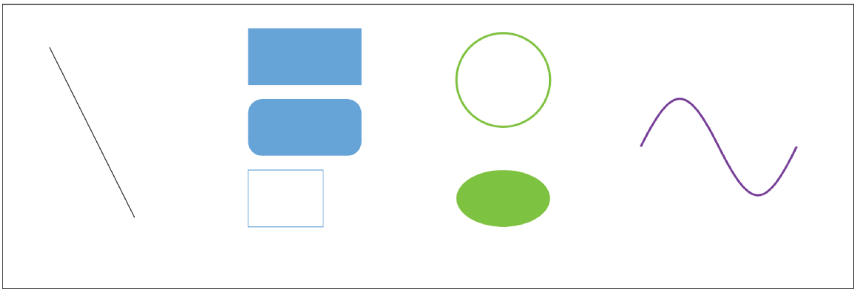


Рис. 15.8 Мы нарисовали все фигуры, какие собирались нарисовать

15.2.6 Текст

Для отображения текста в Canvas существует несколько дополнительных команд. Первая – `fillText()`. Она принимает сам текст и его координаты. В листинге 15.9 мы задаем цвет текста с помощью свойства `fillStyle` контекста, а также размер и семейство шрифта с помощью свойства `font`.

По умолчанию текст выравнивается по левому краю. Задать другой способ выравнивания можно с помощью свойства `textAlign` контекста.

Листинг 15.9 Добавление текста (main.js)

```
context.fillStyle = "#636466";
context.font = "16px monospace";
context.fillText("прямая линия", 80, 346);
context.fillText("прямоугольник", 346, 346);
context.fillText("кривая", 973, 346);
```

Задать цвет, размер и семейство шрифта

Командой fillText() вывести текст в указанное местоположение

```
context.textAlign = "center";
context.fillText("круг", 706, 206);
context.fillText("эллипс", 706, 346);
```

Добавить еще две метки, отцентрированные по горизонтали вокруг заданных координат

На этом мы завершаем создание галереи фигур, показанной на рис. 15.9! Как видите, рисовать фигуры в Canvas довольно просто, но для этого требуется написать чуть больше кода, чем при создании графики SVG. Кроме того, мы не используем D3 для добавления фигур в DOM, как это делается при работе с SVG, за исключением самого элемента `canvas`. Вместо этого мы используем команды рисования Canvas API.

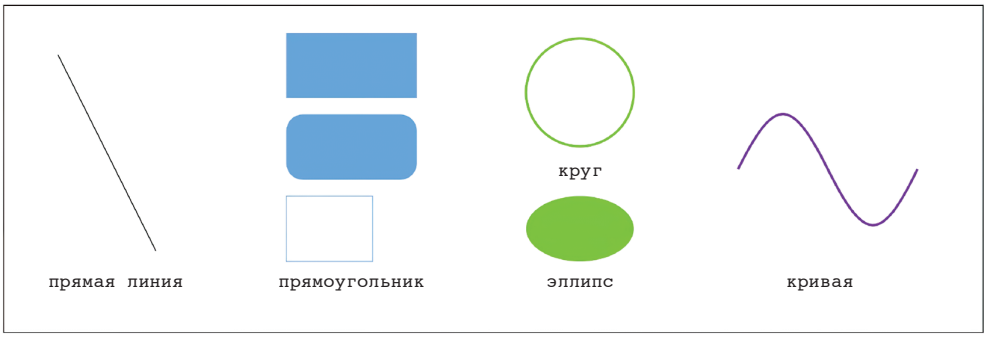


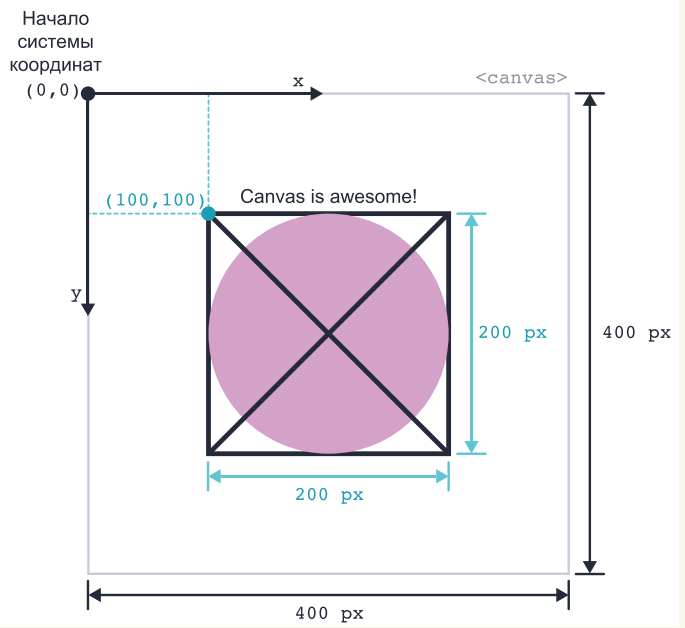
Рис. 15.9 Завершенная галерея фигур

Упражнение: создание графики с помощью Canvas

Теперь ваш черед! Создайте графику Canvas, показанную на рисунке ниже. В качестве отправной точки можете использовать файлы в папке `start` внутри `15.2-Exercise` в репозитории этой главы (http://mng.bz/EZED). Вот несколько рекомендаций, которые вам помогут:

1. создайте адаптивный элемент `canvas` с максимальной шириной и высотой 400 пикселей. Если экран меньше 400 пикселей, то элемент холста должен занимать всю ширину экрана. Ширина и высота всегда должны быть одинаковыми;
2. вы можете нарисовать серую границу вокруг элемента `canvas`, чтобы видеть пространство, в котором работаете;
3. нарисуйте квадрат с шириной и высотой 200 пикселей. Поместите его в центр контейнера `canvas`, задайте прозрачную заливку и черную обводку толщиной 3 пикселя;
4. добавьте в центр круг с радиусом 100 пикселей. Задайте в его атрибуте `fill` CSS-цвет `"plum"`;
5. нарисуйте две черные диагонали толщиной в 3 пикселя: одну из верхнего левого угла квадрата в нижний правый угол, а другую из верхнего правого угла в нижний левый;

6 добавьте текст "Canvas is awesome!" (Canvas великолепен!) над квадратом и расположите его по центру canvas. Установите следующие свойства стиля для текста: размер шрифта 18 пикселей и семейство шрифтов "sans-serif".



Мы рекомендуем создать эту графику, чтобы закрепить знания, полученные в данном разделе

Решение вы найдете в листингах D.26 и D.27 и в папке 15.2-Exercise/end в репозитории этой главы. Но, как обычно, мы призываем вас попытаться выполнить упражнение самостоятельно.

15.3 Смешанный режим отображения

До сих пор в этой главе мы сравнивали Canvas с SVG. Вы можете реализовать проект с использованием только SVG или только Canvas, но есть еще один подход, называемый смешанным, применяя который, можно отобразить часть визуализации с помощью SVG, а часть – с помощью Canvas. Давайте посмотрим, как это можно сделать, взяв в качестве примера наш проект с картинами Ван Гога из главы 14.

Этот проект состоит из серии небольших визуализаций, каждая из которых представляет один год и визуализирует картины, рисунки и письма, написанные и нарисованные художником в течение

этого года. Пример визуализации показан на рис. 15.10. Завершенный проект вы можете найти по адресу http://mng.bz/z8rg. В каждой визуализации количество рисунков, созданных в каждом месяце, отображается с помощью радиальной диаграммы областей, а количество писем – с помощью радиальной столбчатой диаграммы. Каждая картина, написанная Ван Гогом, представлена кругом, размер которого пропорционален размерам картины, а цвет определяет жанр (портрет, пейзаж, натюрморт и т. д.).

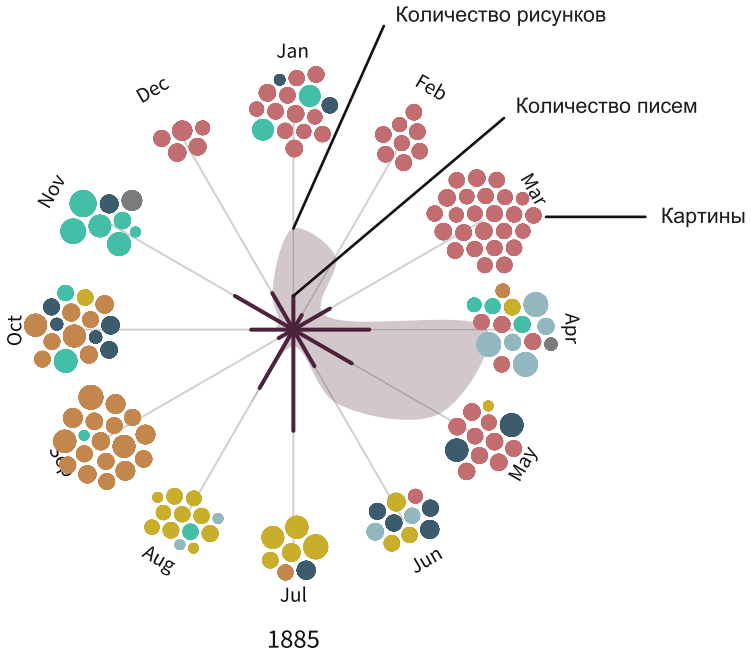


Рис. 15.10 Проект состоит из серии небольших визуализаций, каждая из которых представляет один год творческой жизни Ван Гога. Количество рисунков, созданных в каждом месяце, отображается с помощью радиальной диаграммы областей, а количество писем – с помощью радиальной столбчатой диаграммы. Каждая картина представлена кругом, размер которого пропорционален размерам самой картины, а цвет определяет ее жанр

Учитывая, что в течение 10 лет, которые мы представляем в нашей визуализации, Ван Гог написал более 1000 картин, отображение каждой из них с помощью круга SVG, как это было сделано в главе 14, может создать существенную нагрузку на браузер. Это не особенно критично, потому что на момент написания книги Chrome иногда подвисал, выполняя расчеты для силовой компоновки кругов, а Firefox обычно справлялся с вычислениями вполне нормально (у вас ситуация может отличаться, в зависимости от версии вашего браузера и быстродействия компьютера).

Эти круги – отличный пример визуализации, отображение которой можно реализовать с помощью Canvas. Используя смешанный режим, мы отображаем визуализацию в виде нескольких слоев. В зависимости от желаемого результата и предполагаемых взаимодействий мы должны решить, в каком порядке добавлять эти слои на страницу. В данном проекте нам потребуется всего два слоя: первый слой – графика SVG с осями и диаграммами, и второй – элемент Canvas с кругами, представляющими картины, как показано на рис. 15.11. Размещая элемент холста поверх контейнера SVG, мы гарантируем, что круги картин будут отображаться поверх других визуальных элементов.

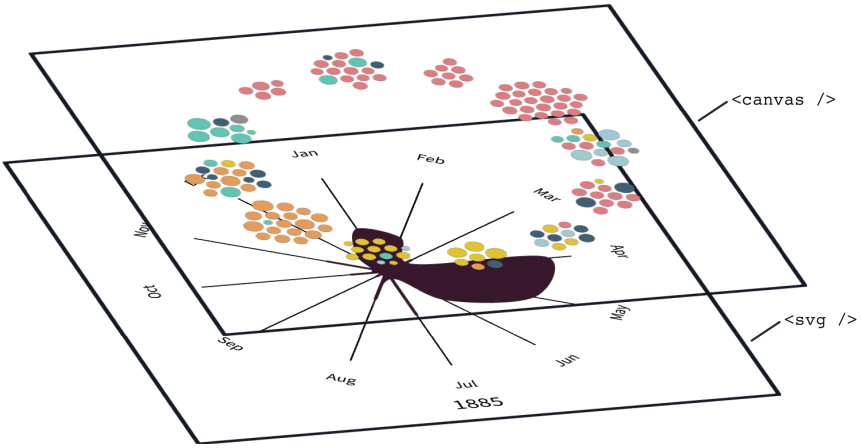


Рис. 15.11 В смешанном режиме отображения мы используем контейнер SVG для отображения осей и диаграмм, отображающих количество рисунков и писем, и поверх него добавляем элемент canvas с кругами, представляющими картины

Чтобы следовать за примерами в этом разделе, откройте папку `start`, которую можно скачать из репозитория главы (http://mng.bz/NRnx) в редакторе кода. Этот проект создан с помощью фреймворка Svelte, краткое описание которого вы найдете в главе 14. Чтобы запустить проект локально, выполните в окне терминала команду `npm install`, а затем `npm run dev`. Адрес и номер порта появятся в окне терминала (обычно это `http://localhost :5173/`).

В листинге 15.10 показан код из файла `Paintings.svelte` (вы найдете его в папке `chart_components`). Мы удалили код, создающий элементы SVG, который был показан в главе 14, и оставили только цветовую шкалу и реализацию силовой компоновки. Цветовая шкала отвечает за определение цвета каждого круга, а реализация силовой компоновки вычисляет их позиции. Некоторые детали силовой компоновки в D3 объясняются в главах 11 и 14, поэтому мы не будем

повторять их здесь. Если вы незнакомы с Svelte, то краткое введение вы найдете в приложении E.

Листинг 15.10 Отправная точка (Paintings.svelte)

```
<script>
 import { forceSimulation, forceX, forceY,
   forceCollide }
   ➥ from "d3-force";
 import { scaleOrdinal } from "d3-scale";
 import { subjects } from "../utils/subjects";
 export let paintingAreaScale;
 export let paintingDefaultRadius;
 export let monthScale;
 export let radius;
 export let isTooltipVisible = false;
 export let tooltipMeta = {};

 export let width;
 export let height;
 export let paintings;
 export let yearsTranslations;

 const colorScale = scaleOrdinal()
   .domain(subjects.map((d) => d.subject))
   .range(subjects.map((d) => d.color));

 let simulation = forceSimulation(paintings);
 let nodes = [];
 simulation.on("tick", () => {
   nodes = simulation.nodes();
 });

 $: {
   simulation
     .force(
       "x",
       forceX((d) => {
        const translation = yearsTranslations.find(
          (y) => y.year === d.year
        ).translationX;
        return d.month !== ""
          ? translation + radius * Math.sin(
              monthScale(d.month))
          : translation;
      }).strength(0.5)
     )
     .force(
       "y",
       forceY((d) => {
```

Импортировать зависимости

Импортировать свойства из родительского компонента

Инициализировать переменные

Объявить порядковую шкалу для получения цвета, соответствующего жанрам картин

Инициализировать массив узлов и смоделировать силовую компоновку. После каждой итерации вызвать метод nodes() модели

Применить силы позиционирования и столкновения

```
        const translation = yearsTranslations.find(
          (y) => y.year === d.year
        ).translationY;
        return d.month !== ""
          ? translation - radius * Math.cos(
              monthScale(d.month))
          : translation;
      }).strength(0.5)
    )
    .force(
      "collide",
      forceCollide()
        .radius((d) =>
          d.width_cm === null && d.height_cm === null
            ? paintingDefaultRadius + 1
            : Math.sqrt(paintingAreaScale(d.area_cm2)
                / Math.PI) + 1
        )
        .strength(1)
    )
    .alpha(0.5)
    .alphaDecay(0.1);
  }
</script>
```

Применить силы позиционирования и столкновения

Открыв дерево DOM проекта в инспекторе, вы обнаружите, что визуализация отображается в контейнере SVG. Согласно стратегии, проиллюстрированной на рис. 15.11, мы должны добавить элемент `canvas` поверх него. В листинге 15.11 мы добавляем элемент `canvas` в разметку компонента и присваиваем его атрибутам `width` и `height` соответствующие значения, полученные в виде входных свойств. Они совпадают с шириной и высотой контейнера SVG и обновляются динамически при изменении размера экрана. Поэтому элемент `canvas` всегда будет иметь точно такой же размер, как и контейнер SVG. Мы также корректируем эти атрибуты с помощью свойства `devicePixelRatio` объекта `window`, чтобы обеспечить достаточное разрешение, как обсуждалось в разделе 15.2.1.

Наконец, мы используем прием абсолютного позиционирования, чтобы поместить элемент `canvas` поверх контейнера SVG. Обратите внимание, что родительский `<div>` контейнера SVG и элемент `canvas` уже имеют CSS-свойство `position`, настроенное со значением `relative`, что обеспечивает правильное выравнивание.

Листинг 15.11 Добавление элемента canvas (Paintings.svelte)

```
<script>
  ...
</script>
```

```
<canvas
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
/>

<style>
  canvas {
    position: absolute;
    top: 0;
    left: 0;
    max-width: 100%;
  }
</style>
```

Добавить элемент canvas в DOM и задать его атрибуты width и height, присвоив им значения входных свойств width и height. Скорректировать их с помощью свойства devicePixelRatio объекта window, чтобы обеспечить достаточное разрешение

Использовать абсолютное позиционирование CSS, чтобы поместить элемент canvas над контейнером SVG

Вернувшись в окно инспектора в браузере, вы увидите, что элемент `canvas` был добавлен на одном уровне с элементом контейнера SVG и отображается поверх него, как показано на рис. 15.12.



Рис. 15.12 Элемент canvas находится поверх контейнера SVG

Теперь нужно инициализировать контекст Canvas. Для этого используем ссылку на узел `canvas`, которая в Svelte доступна в виде привязки `this`, как показано в листинге 15.12. Эта привязка ссылается на переменную `canvasElement`, которая инициализируется выше в тегах `<script>`.

Мы также инициализируем контекст Canvas, вызывая метод `getContext("2d")` элемента canvas. Этот вызов должен производиться внутри метода жизненного цикла `onMount()`, чтобы вызов произошел уже после добавления узла `canvas` в DOM. Наконец, мы применяем горизонтальное и вертикальное масштабирование к единицам холста, чтобы учесть предыдущую коррекцию разрешения.

Обратите внимание, что мы объявили эти переменные до запуска симуляции силовой компоновки, потому что они используются внутри симуляции.

Листинг 15.12 Инициализация контекста Canvas (Paintings.svelte)

```
<script>
  ...

  let simulation = forceSimulation(paintings);
  let nodes = [];
  simulation.on("tick", () => {
    nodes = simulation.nodes();
  });

  let canvasElement;
  let context;
  onMount(() => {
    context = canvasElement.getContext("2d");
    context.scale(window.devicePixelRatio,
      window.devicePixelRatio);
  });
  $: {
    simulation
      .force ...
  }
</script>

<canvas
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={canvasElement}
/>
```

Объявить переменную canvasElement, которая будет привязана к узлу <canvas>

Установить контекст Canvas внутри функции onMount(), чтобы гарантировать, что к этому моменту узел <canvas> уже существует в DOM

Применить преобразование масштаба для учета коррекции разрешения, выполненной ранее

Привязать узел <canvas> к переменной canvasElement

Завершив подготовку Canvas, можно начинать рисовать круги! В листинге 15.13 мы создаем функцию с именем `handleSimulationEnd()`. В симуляцию силовой компоновки мы добавляем обработчик события `"end"`, соответствующего остановке таймера симуляции, который вызывает функцию `handleSimulationEnd()`.

Внутри `handleSimulationEnd()` мы обходим узлы в цикле и рисуем их на холсте один за другим. Поскольку каждый круг имеет цвет, соответствующий жанру картины, и обводку, представляющую технику исполнения, мы устанавливаем свойства `fillStyle` и `strokeStyle`. Затем используем команды, обсуждавшиеся в разделе 15.2.4:

1. начинаем новую линию вызовом `beginPath()`;
2. задаем координаты и размеры круга вызовом `arc(x, y, radius, startAngle, endAngle)`;
3. рисуем заливку и обводку круга с помощью методов `fill()` и `stroke()`.

Листинг 15.13 Рисование окружностей (Paintings.svelte)

```
<script>
  ...

  let simulation = forceSimulation(paintings);
  let nodes = [];
  simulation.on("tick", () => {
    nodes = simulation.nodes();
  });

  let canvasElement;
  let context;
  onMount(() => {
    context = canvasElement.getContext("2d");
    context.scale(window.devicePixelRatio, window.devicePixelRatio);
  });

  const handleSimulationEnd = () => {
    nodes.forEach((node) => {
      context.fillStyle = colorScale(node.subject);
      switch (node.medium) {
        case "oil":
          context.strokeStyle = "#FFFAFC";
          break;
        case "watercolor":
          context.strokeStyle = "#160E13";
          break;
        case "print":
          context.strokeStyle = "#BC5D9A";
          break;
      };

      context.beginPath();
      context.arc(
        node.x,
        node.y,
        node.area_cm2
          ? Math.sqrt(paintingAreaScale(node.area_cm2)
              / Math.PI)
          : paintingDefaultRadius,
        0,
        2 * Math.PI
      );
      context.fill();
      context.stroke();
    });
  };
```

Обойти узлы в цикле

Установить свойства fillStyle и strokeStyle контекста, присвоив им желаемый цвет заливки и обводки круга

Нарисовать круг

```
  $: {
    simulation
      .force ...
      .alpha(0.5)
      .alphaDecay(0.1)
      .on("end", handleSimulationEnd);
  }
</script>

<canvas
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={canvasElement}
/>
```

По завершении симуляции силовой компоновки вызвать функцию handleSimulationEnd(), чтобы нарисовать круги

Вот и все! Круги вернулись на место, но теперь они нарисованы с помощью элемента Canvas (см. рис. 15.13).

15.4 Стратегия взаимодействия с Canvas

Ранее мы упоминали, что обработка взаимодействий с Canvas выглядит сложнее, чем с SVG. Мы не можем полагаться на обычные обработчики событий, как в предыдущих главах, потому что Canvas существует как один элемент DOM, и все, что изображено на нем, недоступно непосредственно. И все же мы не беспомощны.

Во-первых, мы можем обрабатывать события мыши, возникающие на самом элементе Canvas. С этими событиями поставляется множество информации, включая координаты x и y указателя мыши. Если мы визуализируем простые геометрические фигуры и знаем их местоположение, то сможем определить, над какой из этих фигур находится указатель, и отреагировать соответствующим образом. Логика этого подхода относительно проста, но его практическая реализация может быстро стать сложной, особенно при работе со сложными фигурами.

Второй подход немного неожиданный, но весьма остроумный и относительно простой в реализации. Снова используем наш пример с картинами Ван Гога и добавим еще один элемент Canvas поверх созданного в предыдущем разделе, как показано на рис. 15.14. На этом новом элементе Canvas мы снова отобразим те же круги, но на этот раз присвоим каждому кругу случайный и уникальный цвет. По мере перебора картин мы сохраним все их данные и связанные с ними случайные цвета. Затем установим CSS-свойство `opacity` этого элемента Canvas равным `0`. С этого момента мы будем называть его «скрытым» холстом.

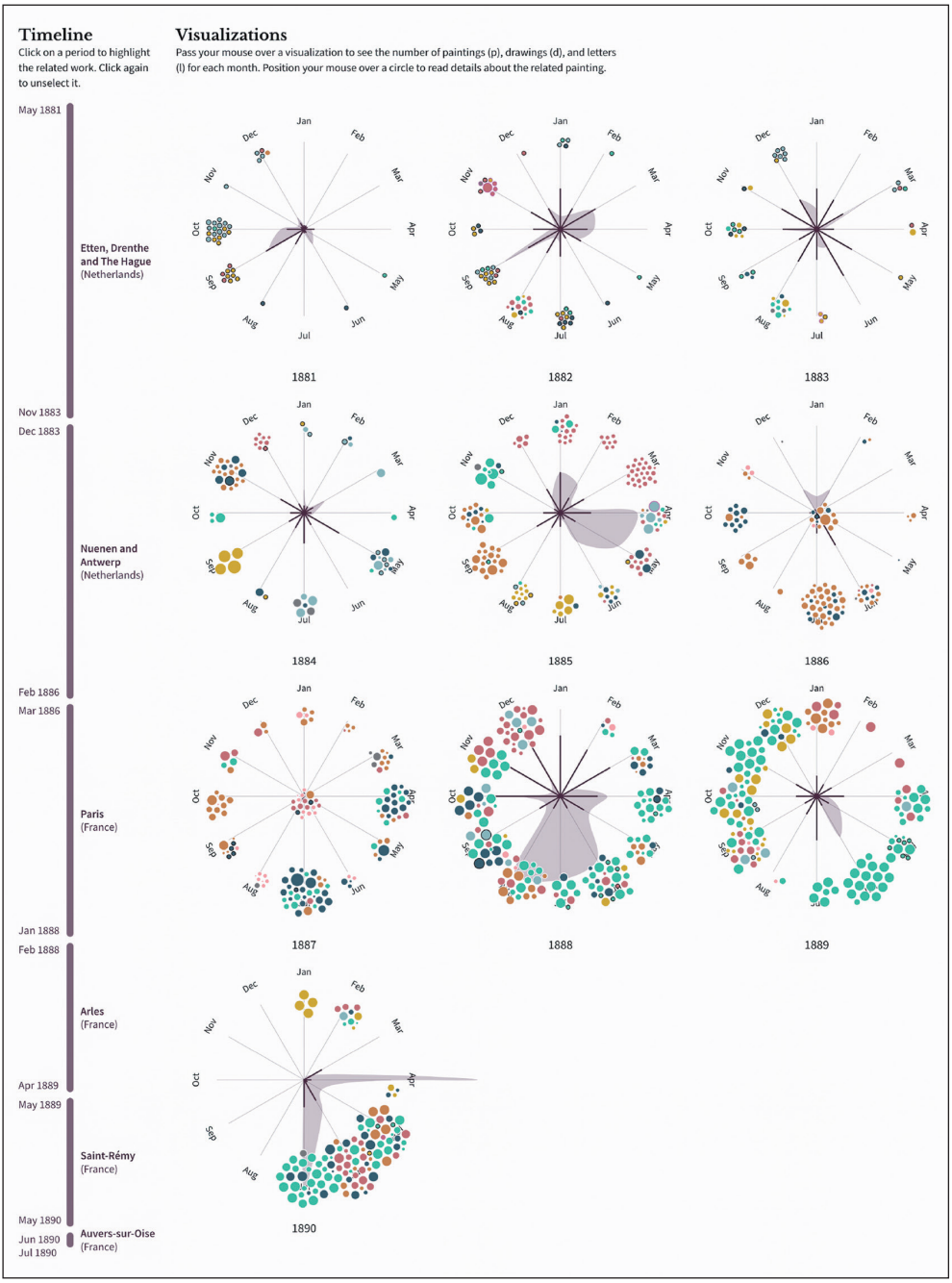


Рис. 15.13 Круги вернулись на место, но теперь они нарисованы с помощью элемента Canvas!

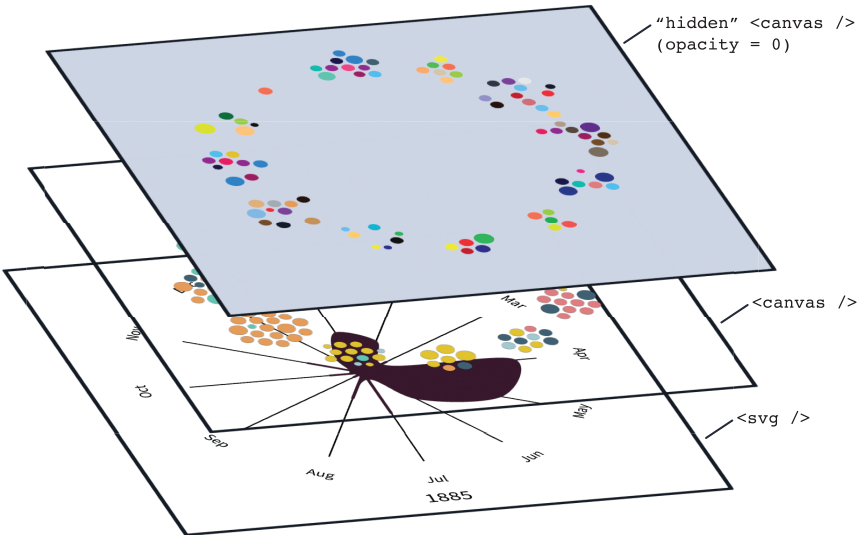


Рис. 15.14 Чтобы определить круг, над которым находится указатель мыши, мы добавляем дополнительный элемент Canvas, в котором каждому кругу присваивается случайный и уникальный цвет. Этот слой скрыт от пользователя установкой его свойства `opacity` в значение 0

Давайте сначала выполним эти шаги, а затем мы продолжим объяснение данного приема. В листинге 15.14 мы добавляем второй элемент Canvas в разметку компонента и привязываем его к новой переменной с именем `hiddenCanvasElement`. Мы также задаем контекст нового элемента холста и сохраняем его в переменной с именем `hiddenContext`. Поскольку этот контекст будет использоваться для обработки частых взаимодействий, его свойству `willReadFrequently` присваивается значение `true`.

Листинг 15.14 Добавление нового элемента Canvas (Paintings.svelte)

```
<script>
  ...

  let canvasElement;
  let context;
  let hiddenCanvasElement;
  let hiddenContext;
  onMount(() => {
    context = canvasElement.getContext("2d");
    context.scale(window.devicePixelRatio, window.devicePixelRatio);
    hiddenContext = hiddenCanvasElement.getContext(
      "2d", { willReadFrequently: true }
    );
```

Объявить переменную hiddenCanvasElement, которая будет привязана к новому узлу <canvas> и его контексту

Инициализировать новый контекст Canvas

```
    hiddenContext.scale(window.devicePixelRatio, window.devicePixelRatio);
  });

  ...
</script>

<canvas
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={canvasElement}
/>
<canvas
  class="hidden-canvas"
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={hiddenCanvasElement}
/>
```

Добавить новый элемент canvas в разметку компонента. Привязать узел к переменной hiddenCanvasElement

Следующий шаг – сгенерировать уникальный цвет для каждого круга, который будет нарисован на новом холсте. Цвет и связанный с ним узел мы сохраним в JavaScript-объекте `Map()`, чтобы потом было проще получать узлы по их цветам. В листинге 15.15 представлены две новые функции. Первая, `generateColor()`, возвращает массив из трех чисел от 0 до 255, а вторая, `addNodeColor()`, собирает эти числа в значение цвета в формате RGB. Если получившийся цвет отсутствует в ассоциативном массиве `nodesColor`, то создается новая пара ключ–значение с цветом RGB и связанным с ним узлом. В `handleSimulationEnd()` мы вызываем `addNodeColor()`, чтобы получить уникальный цвет для каждого круга, представляющего узел, и нарисовать эти круги на невидимом холсте.

Листинг 15.15 Присваивание уникального цвета каждому узлу (Paintings.svelte)

```
<script>
  ...

  let canvasElement;
  let context;
  let hiddenCanvasElement;
  let hiddenContext;
  onMount(() => {
    ...
  });

  const nodesColors = new Map();

  const generateColor = () => {
    let colorArray = [];
    for (let i = 0; i < 3; i++) {
```

Объявить новый ассоциативный массив

Сгенерировать и вернуть массив из трех чисел от 0 до 255

```
      const randomNumber = Math.floor(Math.random()
        * 255);
      colorArray.push(randomNumber);
    }
    return colorArray;
  };

  const addNodeColor = (node) => {
    let isNewGeneratedColor = false;
    while (!isNewGeneratedColor) {
      const colorArray = generateColor();
      const colorRGB = `rgb(${colorArray.join(",")})`;
      if (!nodesColors.get(colorRGB)) {
        nodesColors.set(colorRGB, node);
        isNewGeneratedColor = true;
        return colorRGB;
      }
    }
  };

  const handleSimulationEnd = () => {
    nodes.forEach((node) => {
      ...

      const color = addNodeColor(node);
      hiddenContext.fillStyle = color;
      hiddenContext.strokeStyle = color;

      hiddenContext.beginPath();
      hiddenContext.arc(
        node.x,
        node.y,
        node.area_cm2
          ? Math.sqrt(paintingAreaScale(node.area_cm2)
              / Math.PI)
          : paintingDefaultRadius,
        0,
        2 * Math.PI
      );
      hiddenContext.fill();
      hiddenContext.stroke();
    });
  };

  ...
</script>
```

Сгенерировать и вернуть массив из трех чисел от 0 до 255

Получить новый массив из трех чисел и собрать его в цвете RGB

Если ассоциативный массив nodesColors не содержит ключа, соответствующего вновь созданному цвету RGB, то добавить новую пару ключ–значение с цветом и данными узла. Верните этот цвет

Вызвать addNodeColor(), чтобы получить новый цвет RGB. Присвоить свойствам заливки и обводки контекста этот цвет

Нарисовать новый круг

Если теперь открыть проект в браузере, то вы увидите, что круги на новом холсте Canvas расположены поверх предыдущих и, возможно, имеют очень яркие цвета! Но не волнуйтесь – теперь мы их скроем. В листинге 15.6 мы устанавливаем CSS-свойство `opacity` нового холста в значение `0`.

Листинг 15.16 Сокрытие нового холста (Paintings.svelte)

```
<script>

  ...

</script>

<canvas
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={canvasElement}
/>
<canvas
  class="hidden-canvas"
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={hiddenCanvasElement}
/>

<style>
  canvas {
    position: absolute;
    top: 0;
    left: 0;
    max-width: 100%;
  }
  .hidden-canvas {
    opacity: 0;
  }
</style>
```

У нас все готово к обработке событий мыши, происходящих над кругами. Вот что мы сделаем: Canvas 2D API предоставляет метод `getImageData()`, который принимает координаты точки на холсте и возвращает соответствующий цвет пикселя. Обнаружив событие `mousemove` над элементом canvas, мы используем координаты указателя мыши, поставляемого вместе с событием, и извлекаем соответствующий цвет. Самое интересное, что этот прием работает, даже если элемент canvas невидим. Затем с помощью ассоциативного массива, созданного при присваивании уникального цвета каждому узлу, извлекаем картину, соответствующую кругу, над которым находится указатель мыши. Теперь мы будем знать, когда показывать подсказку и какую информацию она должна включать!

В листинге 15.17 мы добавляем обработчик события `mousemove` в скрытый элемент canvas. Этот обработчик вызывает функцию `han-`

dleMouseMove(), внутри которой мы вызываем метод getImageData() контекста скрытого элемента Canvas. Мы передаем ему прямоугольник для получения данных об изображении. Нам нужно определить цвет в точке, над которой находится указатель мыши, поэтому мы сообщаем, что прямоугольник имеет ширину и высоту 1 пиксель. Поэтому при вызове getImageData() мы передаем четыре аргумента: координаты x и y указателя мыши, а также ширину и высоту, равные 1 пикселю. Обратите внимание, как корректируются координаты x и y, чтобы учесть разрешение. Это необходимо, потому что ранее мы применили преобразование масштаба к Canvas.

Первые три значения в свойстве data соответствуют цвету RGB, так что нам остается только реконструировать цвет RGB, найти соответствующий узел в nodesColor и передать эту информацию компоненту подсказки, созданному в главе 14.

Листинг 15.17 Определение круга, над которым находится указатель мыши (Paintings.svelte)

```
<script>

  ...

  let currentColor = "";
  const handleMouseMove = (event) => {
    const mouseX = event.layerX;
    const mouseY = event.layerY;

    const imageData = hiddenContext.getImageData(
      mouseX * window.devicePixelRatio,
      mouseY * window.devicePixelRatio,
      1,
      1).data;
    const colorRGB = `rgb(
      ${imageData[0]},
      ${imageData[1]},
      ${imageData[2]}
    )`;

    if (colorRGB !== currentColor) {
      if (colorRGB !== "rgb(0,0,0)") {
        currentColor = colorRGB;
        const nodeInfo = nodesColors.get(colorRGB);
        if (nodeInfo) {
```

Получить координаты указателя мыши относительно холста

Извлечь цвет в точке с этими координатами

Найти узел, связанный с этим цветом

```
          isTooltipVisible = true;
          tooltipMeta = {
            x: mouseX,
            y: mouseY,
            screenY: event.clientY,
            url: nodeInfo.imageLink,
            title: nodeInfo.title,
            createdIn: nodeInfo.created_in,
            date:
              nodeInfo.month !== ""
                ? `${nodeInfo.month} ${nodeInfo.year}`
                : nodeInfo.year,
            medium: nodeInfo.medium,
            currentLocation: nodeInfo.current_location,
            width: nodeInfo.width_cm,
            height: nodeInfo.height_cm,
            subject: nodeInfo.subject,
          };
        }
        } else {
          isTooltipVisible = false;
        }
      }
    };

    ...

</script>

<canvas
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={canvasElement}
/>
<canvas
  class="hidden-canvas"
  width={width * window.devicePixelRatio}
  height={height * window.devicePixelRatio}
  bind:this={hiddenCanvasElement}
  on:mousemove={handleMouseMove}
/>
```

Передать данные в подсказку и сделать ее видимой

Подключить обработчик событий к скрытому холсту

Теперь подсказка (см. рис. 15.15) снова работает! Как видите, обработка взаимодействий с визуализациями, построенными на основе Canvas, сложнее, чем в SVG. Тем не менее использование метода `getImageData()` из Canvas API позволило нам обойти эти ограничения.

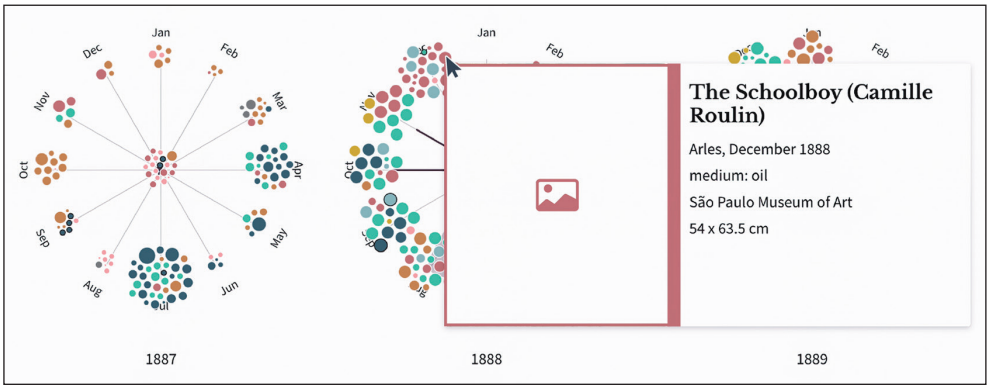


Рис. 15.15 При работе с Canvas мы должны использовать неординарные стратегии для обработки взаимодействий с пользователем. В этом проекте мы придали уникальные цвета кругам картин, нарисованным на скрытом элементе Canvas, извлекли эти цвета в обработчике событий мыши с помощью метода `getImageData()` и получили соответствующие им картины

Интервью с Джулианом Феллом

Фелл – журналист и разработчик.

Расскажите немного о себе и о том, как вы пришли в сферу визуализации данных.

Я работал в сфере разработки ПО около шести лет, занимаясь разнообразными задачами, поэтому в итоге у меня появился довольно широкий набор навыков. Когда я работал инженером в ABC (Australian Broadcasting Corporation), у меня появилась возможность пополнить ряды специалистов по визуализации данных. Эта история была на самом деле моей первой в тот период, и на ее успех немало повлиял вклад опытных членов команды.

Используете ли вы другие инструменты визуализации, помимо D3? Какое место D3 занимает в вашем рабочем процессе визуализации данных?

В основном я использую комбинацию Svelte, D3 и CSS. Мощные средства обработки данных, имеющиеся в D3, можно напрямую использовать для включения декларативной разметки SVG в компоненты Svelte. Фреймворк Svelte поддерживает создание легковесных компонентов (некоторые компоненты, такие как оси, часто можно использовать повторно, но большинство компонентов уникальны для приложения). Он также предоставляет утилиты анимации и управления состоянием, а в D3 имеются надежные утилиты, упрощающие обработку данных.

Один из инструментов, который мне очень нравится (но я не использовал его в этом проекте), – это Layer Cake (https://layercake.graphics/). Он добавляет полезную структуру для построения диаграмм с помощью этих инструментов.

Представленный проект: «How the Seeds of the 2022 Election Result Were Sown Years Ago» (http://mng.bz/Ddzg)

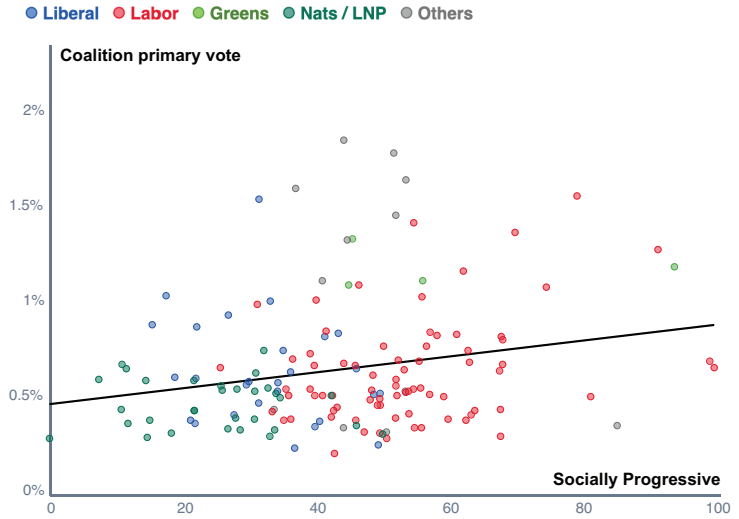


Диаграмма рассеяния, созданная с помощью построителя, написанного Феллом, и похожая на ту, что опубликована в статье

Опишите, пожалуйста, ваш проект.

В ту пору наша команда работала над серией статей, объясняющих механику федеральных выборов 2022 года в Австралии. В этой статье мы хотели объяснить тенденции, которые могли повлиять на результат. Для этого мы использовали диаграммы рассеяния, чтобы показать корреляции между количеством голосов и демографическими данными в каждом из 151 избирательного округа Австралии.

Мы обнаружили несколько закономерностей – связанных с климатом, экономикой, достатком жителей, географией, происхождением, – которые сохранились с предыдущих выборов. Эти тенденции и два аутсайдера, которые вырвались вперед по сравнению с другими местами в 2019 году, объединились, дали возможное объяснение, почему так много традиционно «надежных» мест было потеряно на выборах.

Расскажите, как создавался этот проект.

Имея достаточно времени, мы стараемся подготовить как можно больше кода и данных. Поэтому я собрал довольно много данных об электорате (большую часть из Бюро статистики Австралии). Внешний вид был разработан моим коллегой Беном Спраггоном (Ben Spraggon).

Инженерная группа ABC предоставила API, возвращающий результаты в реальном времени, с которым мы тоже смогли интегрироваться заранее. Затем, используя Svelte и D3, я создал универсальный компонент диаграммы рассеяния. Этот компонент был, пожалуй, самой простой и единственной видимой частью проекта – яркий показатель, что выбор правильных данных и гибкость визуализации являются ключом к рассказу убедительной истории.

Наконец, я создал то, что мы называем интерфейсом «построителя» (http://mng.bz/v8xq). Он позволяет настраивать состояние диаграммы: используемые наборы данных, метки осей, переключатели линий трендов и выбор цветов. Все эти настройки затем были добавлены в визуализацию.

Мне понравилось, что формат прокручиваемой истории, выбранный для этой статьи, позволяет легко следить за повествованием. Визуализация данных подкрепляет понимание читателя. Видите ли вы как журналист существенную разницу между написанием статей с интерактивными визуализациями данных и без них?

Конечно! Сначала я пишу максимально краткий текст статьи, сосредоточиваясь в основном на потоке и порядке визуальных элементов. Затем, разместив диаграммы в правильном порядке, я дополняю статью деталями. Это означает, что на самом деле история рассказывается данными. Последний этап – это тонкая корректировка скорости повествования. При использовании диаграммы, которую сложно интерпретировать, очень важно детально объяснить ее. Значительная часть этой истории на самом деле просто объясняет, как читать диаграмму рассеяния, чтобы читатель мог раскрыть все интересные закономерности.

Как вам удается работать в новостной организации, где события по определению развиваются стремительно, а также создавать интерактивные истории, на которые требуется время?

Нашей команде повезло, что у нас есть время на подготовку к таким важным событиям, как выборы, поэтому бóльшая часть проектирования, программирования и сбора данных выполняется заранее. Высококачественная визуализация данных требует времени, и это легко заметить. Кое-где это считается неправильным, тем не менее я часто копирую и вставляю целые компоненты из старых проектов, чтобы использовать их в качестве основы, а затем настраиваю их. После настройки каждого аспекта каждой диаграммы они упаковываются в повторно используемые компоненты.

Над какими проектами вы планируете работать в ближайшем будущем?

В последнее время я искал способы объединения визуализаций данных и историй в тематические статьи. Эти два стиля передачи информации дополняют друг друга, и я думаю, что числа и данные усиливают контекстуализацию человеческих историй.

Итоги

- SVG и Canvas – это две технологии, используемые для рисования в веб-браузерах. SVG создает векторную графику, а Canvas – растровую.
- Canvas выбирается в основном из соображений производительности, когда визуализация содержит более тысячи элементов, сложную анимацию и/или сложные цветовые градиенты.
- Контекст отображения Canvas предоставляет множество свойств и методов для рисования.
- Мы не обязаны использовать в своих проектах только SVG или только Canvas. Есть возможность использовать смешанный подход: накладывать элементы SVG и canvas друг на друга, чтобы создать желаемый эффект и получить лучше из обоих миров.
- К элементам SVG можно подключать отдельные обработчики событий, но рисунки, отображаемые в Canvas, недоступны в DOM.
- Чтобы позволить пользователю взаимодействовать с визуализациями, отображаемыми в Canvas, можно использовать разные стратегии. Одна из них заключается в применении уникальных цветов к скрытому элементу Canvas и связывании этих цветов с данными. После этого с помощью метода `getImageData()` из Canvas API можно извлекать эти цвета и связанные с ними данные.

Приложение A
Настройка локальной среды разработки

A.1 Visual Studio Code

Если на вашем компьютере еще не установлен редактор Visual Studio Code (VS Code), то выполните следующие действия.

1. Перейдите по ссылке https://code.visualstudio.com/Download.
2. Выберите версию, соответствующую вашей операционной системе (Windows, Linux или Mac), и загрузите пакет для установки.

Windows

1. Если вы загрузили VS Code для Windows, запустите установщик и следуйте его инструкциям.
2. По завершении установки установите флажок **Launch Visual Studio Code** (Запустить Visual Studio Code) и нажмите кнопку **Finish** (Готово).

Mac OS

1. Если вы работаете с версией для Mac, то найдите и дважды щелкните на загруженном zip-файле, чтобы распаковать его.
2. Перетащите распакованный файл с именем `Visual Studio Code.app` в папку `Applications` (Приложения).
3. Дважды щелкните на значке приложения, чтобы открыть VS Code.

Linux

1. Если вы загрузили VS Code для Linux, то стратегия установки будет зависеть от конкретного дистрибутива Linux. Перейдите по ссылке https://code.visualstudio.com/docs/setup/linux, где вы найдете подробные инструкции.
2. Закончив, откройте VS Code.

A.2 Установка и использование расширения Live Server

Для VS Code существует большое количество расширений, которые можно отыскивать и устанавливать прямо из окна редактора. Одно из этих расширений, Live Server, позволяет запускать локальный веб-сервер одним нажатием кнопки. Оно также имеет функцию автоматической перезагрузки страницы, что довольно удобно. Выполните следующие шаги, чтобы установить Live Server.

1. В VS Code найдите значок **Extensions** (Расширения) в левой панели, как показано на рис. А.1.



Рис. А.1 Значок Extensions (Расширения) в интерфейсе VS Code

2. Щелкните на значке, чтобы открыть панель поиска расширений.
3. В строке поиска введите `live server`.
4. Расширение Live Server, скорее всего, появится в самом начале списка с результатами, как на рис. А.2. Выберите его, чтобы открыть страницу сведений о расширении.

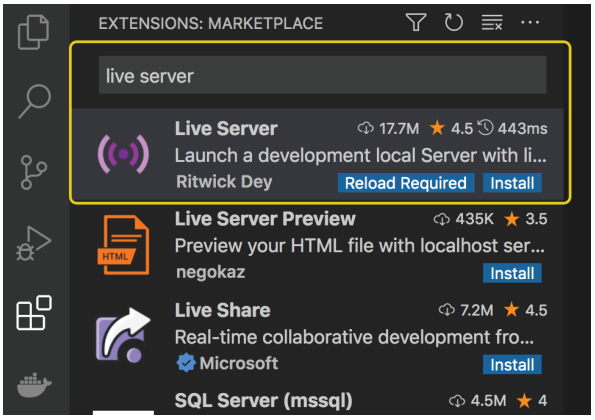


Рис. А.2 Поиск расширения Live Server

5 На странице сведений щелкните на кнопке **Install** (Установить), как показано на рис. А.3.

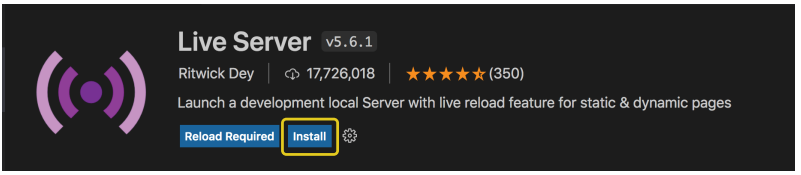


Рис. А.3 Установка расширения Live Server

6 Перезапустите VS Code. Закройте приложение и снова запустите его.

7 Откройте любой веб-проект в VS Code, например папку `start` из репозитория главы 2 (http://mng.bz/Xqjv):
   – для этого откройте окно VS Code, выберите в меню пункт **File > Open** (Файл > Открыть) и перейдите к папке `start`. Если вы работаете на Mac, то вы можете также найти папку с помощью **Finder** и перетащить ее на значок в VS Code.

8 Всякий раз, когда расширение Live Server обнаруживает в папке веб-проект, то делает доступным переключатель **Go Live** в строке состояния в нижней части окна VS Code, как на рис. А.4. Щелкните по нему, чтобы запустить локальный веб-сервер.



Рис. А.4 Щелкните на кнопке Go Live, чтобы запустить локальный веб-сервер

9 Проект автоматически откроется в новой вкладке в вашем браузере по умолчанию. На данный момент это простая белая страница с заголовком «You are about to start working with D3!» («Вы начинаете работу с D3!»), как на рис. А.5. Если новая вкладка не открывается автоматически, то см. шаг 10.

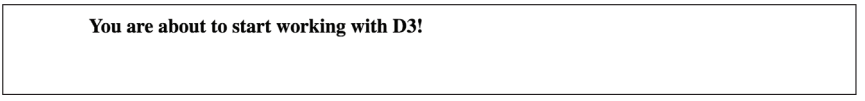


Рис. А.5 Ваш проект автоматически откроется как новая вкладка в браузере по умолчанию

10 Обратите внимание, что в строке состояния надпись «Go Live» на кнопке заменилась номером порта, на котором запущен проект, например **Port: 5500** на рис. А.6. Вы можете вручную

открыть проект в браузере, введя адрес `localhost:port_number` (например, `localhost:5500`).



Рис. А.6 Порт, на котором запущен проект, отображается в строке состояния. Щелкните на нем, чтобы остановить веб-сервер

11 Чтобы протестировать функцию автоматической перезагрузки, измените текст заголовка `h1` в файле `index.html` и сохраните проект. Он автоматически перезагрузит страницу в браузере и покажет ваши изменения.

12 Закончив работу над проектом, вы можете остановить сервер, щелкнув на номере порта в строке состояния.

Приложение B
Выбор шкалы

Цель этого приложения – помочь вам выбрать правильные шкалы D3 для ваших проектов визуализации. За дополнительной информацией обращайтесь к документации модуля d3-scale (https://d3js.org/d3-scale).

Шкалы – это просто функции. Значения, которые они принимают в качестве входных данных, являются частью того, что мы называем *предметной областью* (спектром возможных значений в исходных данных), и возвращают значения, которые являются частью *диапазона* (спектра возможных выходных значений). Предметная область и диапазон могут состоять из непрерывных или дискретных значений. *Непрерывные* значения могут существовать в любом месте в пределах заранее определенного спектра, например число с плавающей точкой от 0 до 100 или дата между июнем 2020 года и январем 2021 года. *Дискретные* значения, напротив, имеют заранее определенный набор значений, например коллекция футболок в размерах XS, S, M, L и XL или группа цветов, таких как синий, зеленый, желтый и красный.

Существует четыре семейства шкал (все они представлены в главе 3). Они отличаются типом значений (непрерывные или дискретные), которые они принимают и возвращают. Для начала выберите семейство из следующего списка и перейдите в указанный раздел:

- непрерывный входной диапазон, непрерывный выходной диапазон: раздел B.1;
- непрерывный входной диапазон, дискретный выходной диапазон: раздел B.2;
- дискретный входной диапазон, непрерывный выходной диапазон: раздел B.3;
- дискретный входной диапазон, дискретный выходной диапазон: раздел B.4.

B.1 Непрерывный входной диапазон, непрерывный выходной диапазон

Первое семейство шкал можно разделить на три категории: непрерывные, последовательные и расходящиеся. Наибольшую практическую ценность для нас имеют непрерывные шкалы, а последовательные и расходящиеся шкалы используются в основном для отображения значений в непрерывном цветовом спектре.

Шкалы этого семейства поддерживают четыре типа преобразований: линейное, степенное, логарифмическое и симметричное логарифмическое. При *линейных преобразованиях* выходное значение линейно пропорционально входному, как показано на рис. B.1.

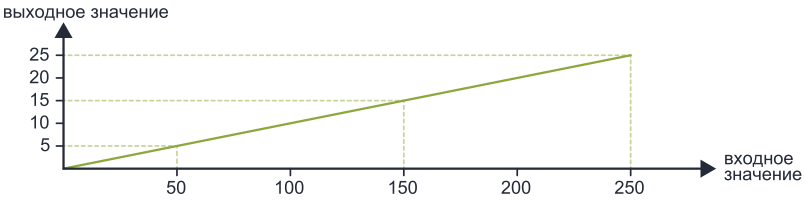


Рис. B.1 При линейном преобразовании выходное значение линейно пропорционально входному

Степенное преобразование применяет экспоненциальную функцию к входным данным. Как показано на рис. B.2, чем больше экспонента, тем круче наклон.

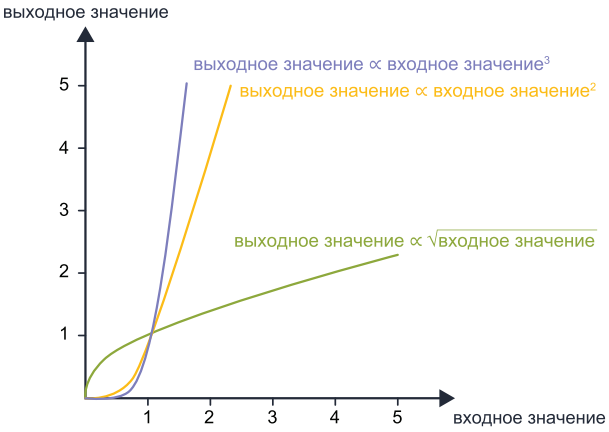


Рис. B.2 Степенные шкалы применяют экспоненциальное преобразование к входным данным

Логарифмическое преобразование позволяет отобразить на одном графике величины, отличающиеся на несколько порядков. При ра-

боте с такими преобразованиями важно помнить, что логарифм нуля не определен, поэтому нулевое значение не должно включаться в предметную область (см. рис. B.3).

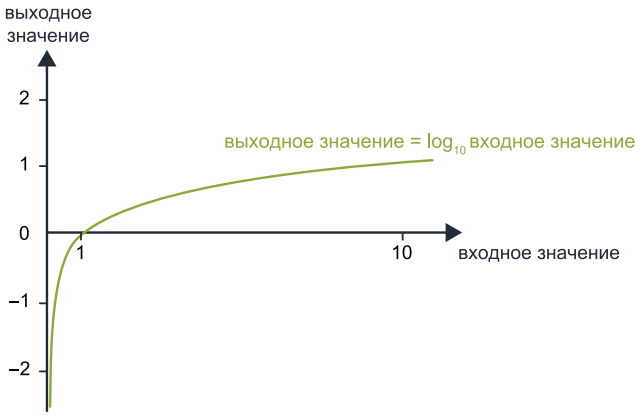


Рис. B.3 Логарифмическое преобразование позволяет отобразить на одном графике величины, отличающиеся на несколько порядков

Однако симметричное логарифмическое преобразование (symlog), показанное на рис. B.4, может включать значение 0.

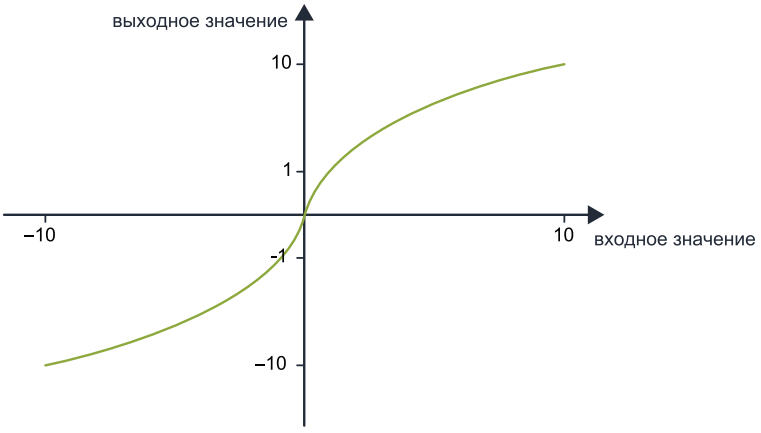


Рис. B.4 В отличие от логарифмической функции, функция симметричного логарифмирования определяет результат для нулевого входного значения

B.1.1 Непрерывные шкалы

В непрерывных шкалах предметная область и выходной диапазон выражаются в виде массивов с двумя значениями: минимума и максимума.

Линейная шкала

Значение, возвращаемое линейной шкалой, прямо пропорционально входному значению, как показано на рис. B.5. Если линейные шкалы не ограничены, они могут экстраполировать значения за пределы заданной предметной области и диапазона.

Метод: `d3.scaleLinear()`.

Преобразование: линейное.

Пример использования: оценки учащихся.

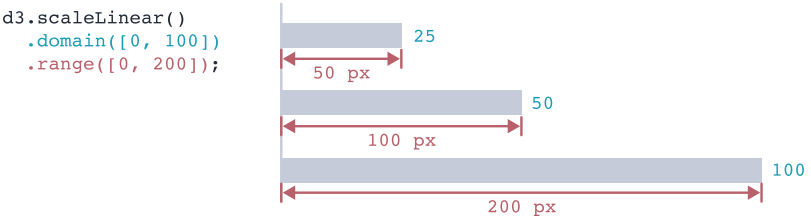


Рис. B.5 Линейная шкала является наиболее часто используемой из всех шкал D3, поскольку она сохраняет пропорциональные различия

Временная шкала

Предметная область временной шкалы выражается как массив из двух JavaScript-объектов `Date()` (выражающих местное время), как на графике на рис. B.6, а диапазон – как массив из двух чисел. Значение, возвращаемое временной шкалой, линейно пропорционально входным данным.

Метод: `d3.scaleTime()`.

Преобразование: линейное.

Пример использования: хронология событий.

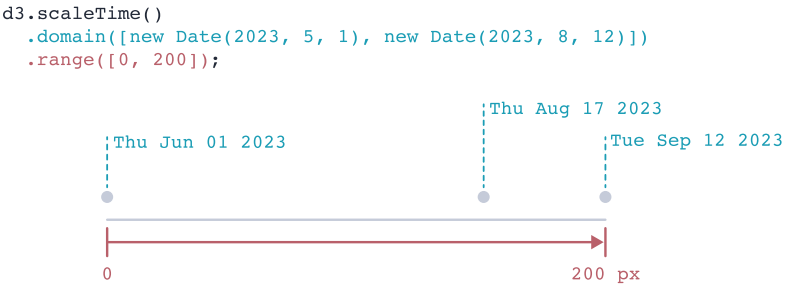


Рис. B.6 Временная шкала принимает объект Date() и возвращает соответствующее числовое значение

Шкала UTC

Шкала UTC подобна временной шкале, но время в ней выражается в формате всемирного координированного времени (Coordinated Universal Time, UTC) – количества миллисекунд, прошедших с 00:00:00 UTC 1 января 1970 года, а не по местному времени, как во временной шкале (см. рис. B.7). Шкала UTC используется, когда важно избежать путаницы с часовыми поясами или летним временем. Для обработки значений дат UTC можно использовать JavaScript-метод `Date.UTC()`.

Метод: `d3.scaleUtc()`.

Преобразование: линейное.

Пример использования: прогноз погоды.

```
const utcDate1 =  new Date(Date.UTC(2023, 5, 1, 11, 45, 5));
const utcDate2 =  new Date(Date.UTC(2023, 8, 12, 11, 45, 5));
d3.scaleUtc()
  .domain([utcDate1, utcDate2])
  .range([0, 200]);
```

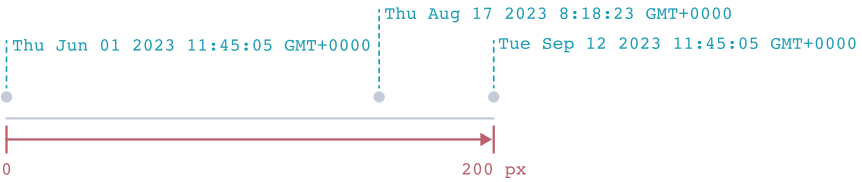


Рис. B.7 Шкала UTC принимает объект Date.UTC() и возвращает соответствующее числовое значение

Шкала идентичности

Шкала идентичности – частный случай линейной шкалы, когда предметная область и диапазон идентичны, как показано на рис. B.8.

Метод: `d3.scaleIdentity()`.

Преобразование: линейное.

Пример использования: иногда полезно при работе с пиксельными координатами в сочетании с методами `d3.zoom()` и `d3.brush`.

```
d3.scaleIdentity()
  .domain([0, 1])
  .range([0, 1]);
```

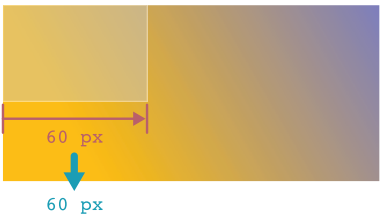


Рис. B.8 Шкала идентичности основана на математической функции идентичности, которая всегда возвращает то же значение, которое получила в виде аргумента

Степенная шкала

Степенная шкала возводит в степень входное значение и возвращает результат, как показано на рис. B.9. Если показатель степени не указан, то по умолчанию он равен 1, что эквивалентно линейной шкале.

Метод: `d3.scalePow().exponent()`.

Преобразование: возведение в степень.

Пример использования: распространение инфекционного заболевания.

```
d3.scalePow()
  .domain([0, 1000])
  .range([0, 200])
  .exponent(2);
```

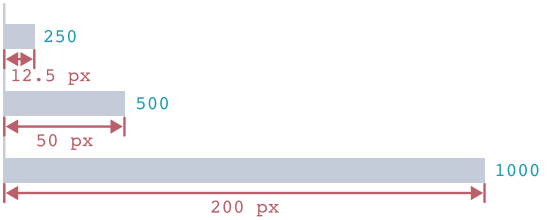


Рис. B.9 Степенная шкала возводит в степень входное значение

Шкала квадратного корня

Шкала квадратного корня – частный случай степенной шкалы с показателем степени 0,5. Эта шкала особенно полезна для круговых визуализаций, как показано на рис. B.10.

Метод: `d3.scaleSqrt()`.

Преобразование: возведение в степень.

Пример использования: радиус окружности.

```
d3.scaleSqrt()
  .domain([0, 1000])
  .range([0, 100]);
```

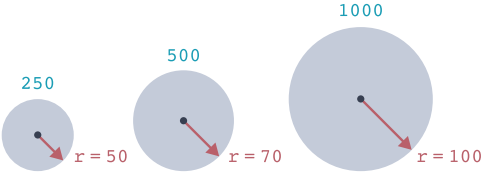


Рис. B.10 Шкалу квадратного корня удобно использовать для расчета радиусов кругов, площадь которых пропорциональна предметному значению

Радиальная шкала

В отличие от степенной шкалы, радиальная шкала возводит входное значение в квадрат. Ее удобно использовать для расчета внутреннего и внешнего радиусов в радиальных диаграммах, как показано на рис. B.11.

Метод: `d3.scaleRadial()`.

Преобразование: возведение в степень.

Пример использования: внутренний и внешний радиусы в радиальных диаграммах.

```
d3.scaleRadial()
  .domain([0, 1000])
  .range([100, 400]);
```

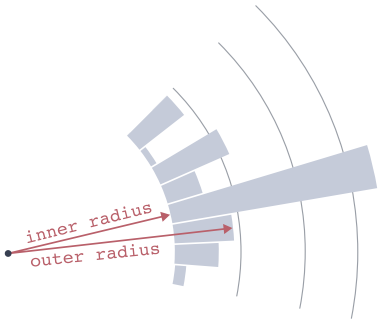


Рис. B.11 Радиальная шкала в основном используется для расчета внутреннего и внешнего радиусов радиальных диаграмм

Логарифмическая шкала

Логарифмическая шкала применяет логарифмическое преобразование к входным данным. Такая шкала позволяет визуализировать несколько порядков величины в пределах одной диаграммы, как показано на рис. B.12. Предметная область должна быть строго положительной или строго отрицательной, потому что логарифма нуля не существует ($\log(0) = -\infty$).

Метод: `d3.scaleLog()`.

Преобразование: логарифмическое.

Пример использования: отображение нескольких порядков величины на одной диаграмме.

```
d3.scaleLog()
  .domain([1, 10M])
  .range([0, 500]);
```

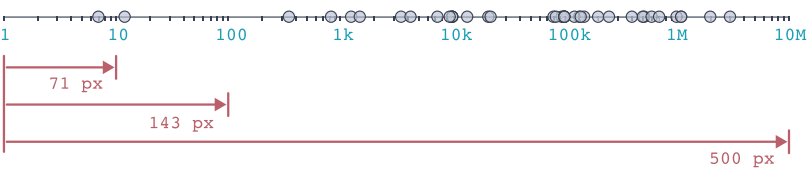


Рис. B.12 Используя логарифмическую шкалу, можно отобразить несколько порядков величины на одной диаграмме

Шкала симметричного логарифмирования

Эта шкала реализует симметричное логарифмическое преобразование (отрицательных и положительных входных значений), как показано на рис. B.13. Предметная область может включать 0.

Метод: `d3.scaleSymlog()`.

Преобразование: симметричное логарифмирование.

Пример использования: как отмечено в документации D3, может применяться для представления времени, чтобы на одной диаграмме показать точки и в близком, и в отдаленном прошлом и будущем.

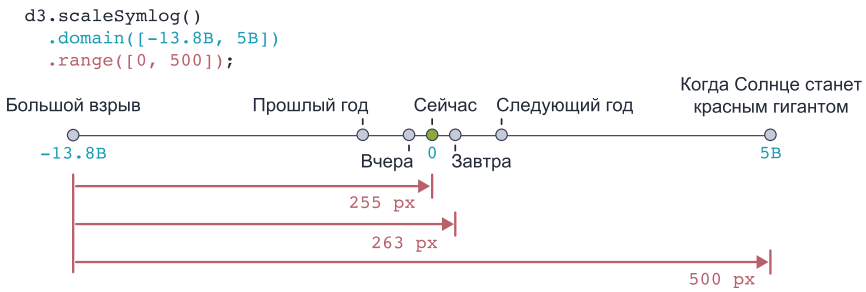


Рис. B.13 Шкала симметричного логарифмирования применяет симметричное преобразование, в котором входной аргумент может принимать значение 0

B.1.2 Последовательные шкалы

Выходное значение последовательных шкал является результатом интерполяции. Эти шкалы обычно используются для отображения непрерывного ввода в непрерывный диапазон цветов.

Последовательная шкала

Результат последовательной шкалы обычно выражается последовательной цветовой схемой, где изменение цвета линейно пропорционально входному значению, как показано на рис. B.14.

Метод: `d3.scaleSequential()`.

Преобразование: линейное.

Пример использования: оценки в виде цвета.

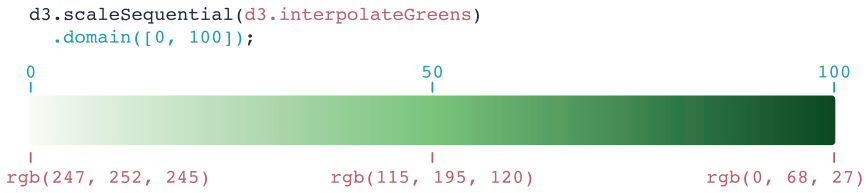


Рис. B.14 Последовательная шкала способна отображать данные в линейный цветовой спектр

Последовательная степенная шкала

Последовательная степенная шкала возвращает результат интерполяции, применяя экспоненциальное преобразование, аналогичное

степенной шкале. Результат применения такой шкалы показан на рис. B.15.

Метод: `d3.scaleSequentialPow().exponent()`.

Преобразование: возведение в степень.

Пример использования: распространение инфекционного заболевания, выраженное в цвете.

```
d3.scaleSequentialPow(d3.interpolateGreens)
  .domain([0, 100])
  .exponent(2);
```



Рис. B.15 Последовательная степенная шкала применяет экспоненциальное преобразование к входным данным

Последовательная шкала квадратного корня

Последовательная шкала квадратного корня – частный случай последовательной степенной шкалы с показателем степени 0,5. Она применяется в тех же случаях преобразования, что и шкала квадратного корня (см. рис. B.16).

Метод: `d3.scaleSequentialSqrt()`.

Преобразование: возведение в степень.

Пример использования: выражение процента доходности как функции риска в финансах.

```
d3.scaleSequentialLog(d3.interpolateGreens)
  .domain([1, 10M]);
```

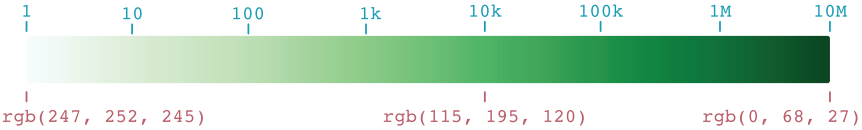


Рис. B.16 Последовательная шкала квадратного корня – частный случай последовательной степенной шкалы с показателем степени 0,5

Последовательная логарифмическая шкала

Последовательная логарифмическая шкала применяет логарифмическое преобразование к входным данным, как показано на рис. B.17. Предметная область должна быть строго положительной или строго отрицательной, потому что логарифма нуля не существует ($\log(0) = -\infty$).

Метод: `d3.scaleSequentialLog()`.

Преобразование: логарифмическое.

Пример использования: отображение нескольких порядков величины в виде цвета.

```
d3.scaleSequentialLog(d3.interpolateGreens)
  .domain([1, 10M]);
```

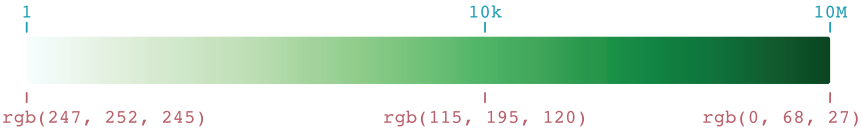


Рис. B.17 Последовательная логарифмическая шкала применяет логарифмическое преобразование

Последовательная шкала симметричного логарифмирования

Последовательная шкала симметричного логарифмирования реализует симметричное логарифмическое преобразование (отрицательных и положительных входных значений), как показано на рис. B.18. Предметная область может включать 0.

Метод: `d3.scaleSequentialSymlog()`.

Преобразование: симметричное логарифмирование.

Пример использования: как отмечено в документации D3, может применяться для представления времени, чтобы на одной диаграмме показать точки и в близком, и в отдаленном прошлом и будущем.

```
d3.scaleSequentialSymlog(d3.interpolateGreens)
  .domain([-10M, 10M]);
```

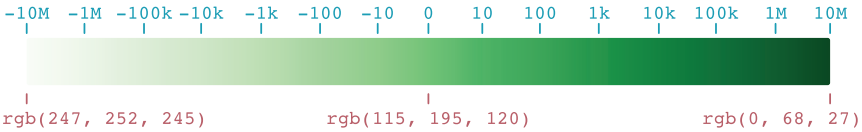


Рис. B.18 Последовательная шкала симметричного логарифмирования применяет симметричное преобразование, придавая бóльшие значения величинам в центре

B.1.3 Расходящиеся шкалы

Расходящиеся шкалы, как и последовательные, возвращают результат применения функции интерполяции. Предметная область расходящихся шкал содержит ровно три значения (минимальное, нейтральное и максимальное). Выходные результаты таких шкал представляют значения (обычно цвета), идущие в противоположных направлениях.

Расходящаяся шкала

Результат применения расходящейся шкалы обычно выражается расходящейся цветовой схемой, где цвет изменяется линейно между тремя значениями, как показано на рис. В.19.

Метод: `d3.scaleDiverging()`.

Преобразование: линейное.

Пример использования: глобальные температурные тенденции в градусах Цельсия.

```
d3.scaleDiverging(d3.interpolatePRGn)
  .domain([-1, 0, 1]);
```



Рис. В.19 Расходящаяся цветовая шкала возвращает непрерывную цветовую схему, направленную в две противоположные стороны

Расходящаяся степенная шкала

Расходящаяся степенная шкала похожа на расходящуюся шкалу, но применяет к входному значению степенное преобразование, как показано на рис. В.20.

Метод: `d3.scaleDivergingPow().exponent()`.

Преобразование: возведение в степень.

Пример использования: распространение инфекционного заболевания, выраженное в цвете.

```
d3.scaleDivergingPow(d3.interpolatePRGn)
  .domain([-1, 0, 1]);
```

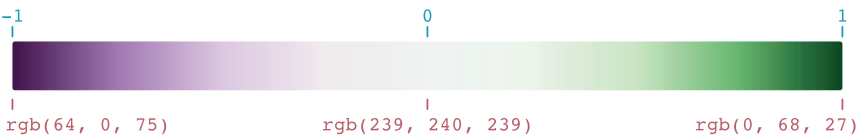


Рис. В.20 Расходящаяся степенная шкала применяет к входному значению степенное преобразование

Расходящаяся шкала квадратного корня

Расходящаяся шкала квадратного корня похожа на расходящуюся степенную шкалу, но с показателем степени 0,5, как показано на рис. В.21.

Метод: `d3.scaleDivergingSqrt()`.

Преобразование: возведение в степень.

Пример использования: выражение процента доходности как функции риска в финансах.

```
d3.scaleDivergingSqrt(d3.interpolatePRGn)
  .domain([-1, 0, 1]);
```

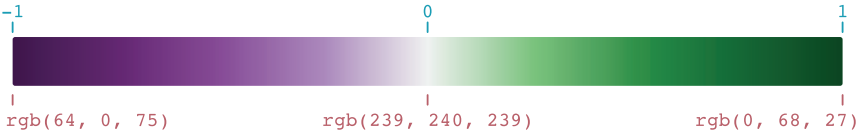


Рис. B.21 Расходящаяся шкала квадратного корня – частный случай расходящейся степенной шкалы с показателем степени 0,5

Расходящаяся логарифмическая шкала

Расходящаяся логарифмическая шкала похожа на расходящуюся шкалу, но применяет к входному значению логарифмическое преобразование. Как показано на рис. B.22, предметная область должна быть строго положительной или строго отрицательной, потому что логарифма нуля не существует ($\log(0) = -\infty$).

Метод: `d3.scaleDivergingLog()`.

Преобразование: логарифмическое.

Пример использования: выражение нескольких порядков величины в виде цвета.

```
d3.scaleDivergingLog(d3.interpolatePRGn)
  .domain([1, 100k, 10M]);
```

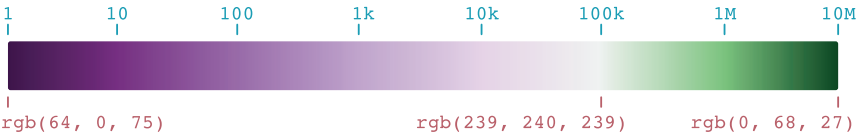


Рис. B.22 Расходящаяся логарифмическая шкала применяет к входному значению логарифмическое преобразование

Расходящаяся шкала симметричного логарифмирования

Эта шкала реализует симметричное логарифмическое преобразование (отрицательных и положительных входных значений), как показано на рис. B.23. Предметная область может включать 0.

Метод: `d3.scaleDivergingSymlog()`.

Преобразование: симметричное логарифмирование.

Пример использования: как отмечено в документации D3, может применяться для представления времени, чтобы на одной диаграмме показать точки и в близком, и в отдаленном прошлом и будущем.

```
d3.scaleDivergingSymlog(d3.interpolatePRGn)
  .domain([-10M, 0, 10M]);
```



Рис. B.23 Расходящаяся шкала симметричного логарифмирования применяет симметричное логарифмическое преобразование, придавая бóльшие значения величинам в центре

B.2 Непрерывный входной диапазон, дискретный выходной диапазон

Второе семейство включает шкалы с непрерывной предметной областью и дискретным диапазоном.

Шкала квантования

Входная предметная область шкалы квантования делится на регулярные интервалы, соответствующие числу дискретных значений в выходном диапазоне, как показано на рис. B.24.

Метод: `d3.scaleQuantize()`.

Пример использования: разделение значений на категории.

```
d3.scaleQuantize()
  .domain([0, 100])
  .range(["yellow", "orange", "green", "purple"]);
```



Рис. B.24 Шкала квантования делит непрерывные значения на дискретные категории

Шкала порогов

Шкала порогов похожа на шкалу квантования, за исключением того, что позволяет решать, на какие сегменты предметной области отображаются дискретные значения из выходного диапазона (см. рис. B.25). Предметная область непрерывна, но задается как набор дискретных значений.

Метод: `d3.scaleThreshold()`.

Пример использования: картограммы.

```
d3.scaleThreshold()
  .domain([5, 7, 12])
  .range(["yellow", "orange", "green", "purple"]);
```

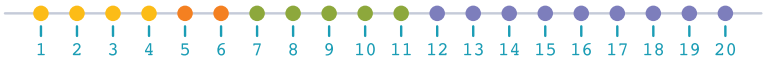


Рис. B.25 Шкала порогов позволяет решать, на какие сегменты предметной области отображаются дискретные значения из выходного диапазона

Шкала квантилей

В шкале квантилей диапазон отображается на квантили выбранной предметной области. Квантили – это группы, содержащие одинаковое количество элементов. Если диапазон содержит четыре цвета, то шкала поделит предметную область на четыре квантиля, и каждому из них назначит свой цвет, как показано на рис. B.26. Предметная область непрерывна, но указывается как набор дискретных значений.

Метод: `d3.scaleQuantile()`.

Пример использования: картограммы.

```
d3.scaleQuantile()
  .domain([1, 2, 5, 6, 8, 9, 10, 11, 12, 18, 19, 20])
  .range(["yellow", "orange", "green", "purple"]);
```

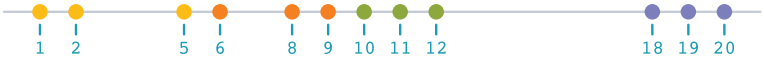


Рис. B.26 Шкала квантилей делит выборку образцов из предметной области на четыре группы, содержащие одинаковое количество элементов, и затем может принимать непрерывные значения из этой области

Последовательная шкала квантилей

Последовательная шкала квантилей сортирует значения из предметной области и сопоставляет их с результатами функции интерполяции для определения размера шага, как показано на рис. B.27.

Метод: `d3.scaleSequentialQuantile()`.

Пример использования: картограммы.

```
d3.scaleSequentialQuantile(d3.interpolatePRGn)
  .domain([0, 2, 5, 9]);
```

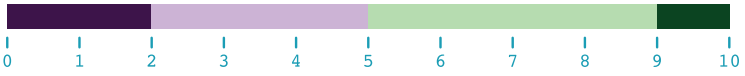


Рис. B.27 Последовательная шкала квантилей сопоставляет квантили с результатами функции интерполяции

B.3 Дискретный входной диапазон, непрерывный выходной диапазон

Третье семейство шкал отображает дискретный входной диапазон в непрерывный выходной диапазон.

Интервальная шкала

Интервальная шкала распределяет дискретные элементы по непрерывному диапазону, назначая каждому элементу из предметной области определенный интервал, исходя из доступного пространства и требуемого заполнения. Эта шкала в основном используется для создания столбчатых диаграмм, как показано на рис. B.28.

Метод: `d3.scaleBand()`.

Пример использования: распределение прямоугольников столбчатой диаграммы вдоль оси.

```
d3.scaleBand()
  .domain(["apple", "apricot", "banana"])
  .range([0, 300])
  .padding(0.1);
```

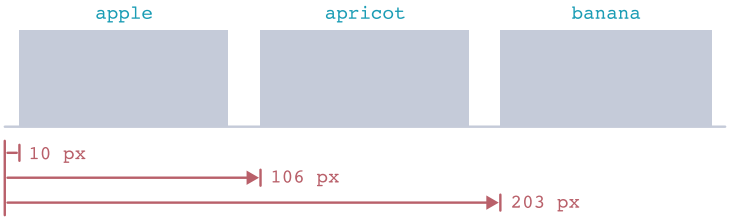


Рис. B.28 Интервальная шкала – предпочтительный инструмент для распределения прямоугольников на гистограмме

Точечная шкала

Точечная шкала похожа на интервальную шкалу, с той лишь разницей, что каждый элемент получает интервал, равный 0, как показано на рис. B.29.

Метод: `d3.scalePoint()`.

Пример использования: диаграммы рассеяния с категориальным измерением.

```
d3.scalePoint()
  .domain(["apple", "apricot", "banana"])
  .range([0, 300]);
```

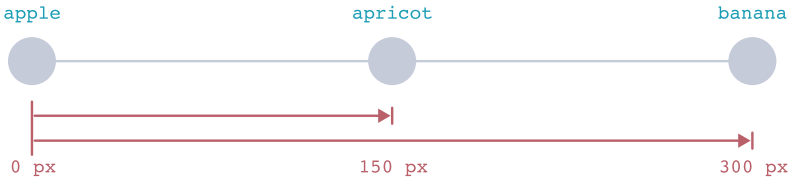


Рис. B.29 Точечная шкала действует подобно интервальной шкале, за исключением того, что каждый элемент получает интервал, равный 0

B.4 Дискретный входной диапазон, дискретный выходной диапазон

Последнее семейство отображает дискретную предметную область в дискретный диапазон.

Порядковая шкала

Порядковая шкала отображает массив элементов из предметной области в другой массив из заданного выходного диапазона. Как показано на рис. B.30, первое значение из предметной области отображается в первое значение из выходного диапазона, второе значение – во второе и т. д. Если число значений в выходном диапазоне меньше, чем в предметной области, то D3 будет циклически перебирать значения из выходного диапазона.

Метод: `d3.scaleOrdinal()`.

Пример использования: сопоставление категорий с набором цветов.

```
d3.scaleOrdinal()
  .domain(["pop", "rock", "hip-hop",  "funk"])
  .range(["yellow", "orange", "green", "purple"]);
```



Рис. B.30 Порядковая шкала отображает дискретную область в дискретный диапазон

Приложение C
Обзор модулей D3

Библиотека D3 разделена на несколько модулей. В этом приложении мы перечислим эти модули и объясним, когда их использовать. Модули, отмеченные звездочкой (*), мы обычно не используем при объединении D3 с JavaScript-фреймворками, такими как React или Svelte, поскольку это может привести к конфликтам.

Таблица C.1 Перечень модулей D3

Предварительная обработка		Рисование элементов визуализации на экране	Заключительные штрихи	
Подготовка данных	Расчет макета		Взаимодействия	Анимация
d3-array	d3-chord	d3-axis*	d3-brush*	d3-ease
d3-color	d3-contour	d3-selection*	d3-dispatch*	d3-timer
d3-dsv	d3-delaunay		d3-drag*	d3-transition*
d3-fetch	d3-force		d3-zoom*	
d3-format	d3-geo			
d3-random	d3-hierarchy			
d3-time	d3-interpolate			
d3-time-format	d3-path			
	d3-polygon			
	d3-quadtree			
	d3-scale			
	d3-scale-chromatic			
	d3-shape			

* Содержит методы, требующие доступа к DOM. Эти модули желательно не использовать при объединении D3 с JavaScript-фреймворком, таким как Reactor Svelte. Объяснение причин и способов обхода вы найдете в главе 8.

Приложение D
Решения упражнений

D.1 Решения для упражнений в главе 1

D.1.1 Построение графики SVG

Листинг D.1 Построение графики SVG (index.html)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>SVG Exercise | D3.js in Action</title>
</head>
<body>
  <div style="width:100%; max-width:400px; margin:0 auto;">
    <svg viewBox="0 0 400 400" style="border:1px solid black;">

      <rect x="100" y="100" width="200" height="200" fill="transparent"
      ➥ stroke="black" stroke-width="5" />

      <circle cx="200" cy="200" r="100" fill="plum" />

      <line x1="100" y1="100" x2="300" y2="300" stroke="black"
      ➥ stroke-width="5" />
      <line x1="100" y1="300" x2="300" y2="100" stroke="black"
      ➥ stroke-width="5" />

      <text x="200" y="90" text-anchor="middle" style="font-size:18px;
      ➥ font-family:sans-serif">SVG is awesome!</text>

    </svg>
  </div>
</body>
</html>
```

D.2 Решения упражнений в главе 6

D.2.1 Построение пирамидальной диаграммы

Листинг D.2 Построение пирамидальной диаграммы (pyramid.js)

```
const dataWomen = data.filter(d =>
  d.gender === "Female");
const binsWomen = d3.bin()
  .value(d => d.salary)(dataWomen);

const dataMen = data.filter(d => d.gender === "Male");
const binsMen = d3.bin()
  .value(d => d.salary)(dataMen);
```

Создать отдельные корзины для заработной платы женщин и мужчин

```
const xScaleWomen = d3.scaleLinear()
  .domain([15, 0])
  .range([0, innerWidth/2]);
const xScaleMen = d3.scaleLinear()
  .domain([0, 15])
  .range([innerWidth/2, innerWidth]);
```

Инициализировать две линейные шкалы для горизонтальной оси: одну для женщин, расположенную вдоль первой половины оси x, и одну для мужчин, расположенную вдоль второй половины

```
const minSalary = binsWomen[0].x0;
const maxSalary = binsWomen[binsWomen.length - 1].x1;
const yScale = d3.scaleLinear()
  .domain([minSalary, maxSalary])
  .range([innerHeight, 0]);
```

Инициализировать линейную шкалу для вертикальной оси, охватывающую диапазон от 0 до максимальной заработной платы

```
const pyramidContainer = innerChart
  .append("g")
    .attr("stroke", white)
    .attr("stroke-width", 2);
```

Добавить группу SVG во внутреннюю диаграмму. Сохранить выборку в константе pyramidContainer и настроить атрибуты stroke и stroke-width группы – они будут наследоваться дочерними элементами этой группы

```
pyramidContainer
  .selectAll(".bar-women")
  .data(binsWomen)
  .join("rect")
    .attr("class", "bar-women")
    .attr("x", d => xScaleWomen(d.length / data.length
      * 100))
    .attr("y", d => yScale(d.x1))
    .attr("width", d => innerWidth/2 -
      xScaleWomen(d.length / data.length * 100))
    .attr("height", d => yScale(d.x0) - yScale(d.x1))
    .attr("fill", womenColor);

pyramidContainer
  .selectAll("bar-men")
  .data(binsMen)
  .join("rect")
```

Используя шаблон привязки данных, добавить прямоугольники SVG, изображающие столбики в гистограммах. Использовать шкалы для расчета местоположений и длин столбцов, причем длины должны быть пропорциональны проценту точек ↓

```
    .attr("class", "bar-men")
    .attr("x", innerWidth/2)
    .attr("y", d => yScale(d.x1))
    .attr("width", d => xScaleMen(d.length /
      data.length * 100) - innerWidth/2)
    .attr("height", d => yScale(d.x0) - yScale(d.x1))
    .attr("fill", menColor);
```

↑ данных, найденных в соответствующей корзине, от общего количества точек данных

```
const bottomAxisFemales = d3.axisBottom(xScaleWomen)
  .tickValues([15, 10, 5, 0])
  .tickSizeOuter(0);
innerChart
  .append("g")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxisFemales);

const bottomAxisMales = d3.axisBottom(xScaleMen)
  .tickValues([5, 10, 15])
  .tickSizeOuter(0);
innerChart
  .append("g")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxisMales);
svg
  .append("text")
    .text("Percent")
    .attr("text-anchor", "middle")
    .attr("x", margin.left + innerWidth/2)
    .attr("y", height - 3);
```

Чтобы нарисовать ось x, задать отдельные генераторы нижней оси для женщин и мужчин и передать соответствующие шкалы. Определить отметки, отображаемые на каждой оси, можно с помощью функции доступа tickValues(). Добавить метку к оси x с текстом «Percent» (Проценты)

```
const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
    .call(leftAxis);
svg
  .append("text")
    .text("Yearly salary (USD)")
    .attr("x", 0)
    .attr("y", 20);
```

Объявить генератор левой оси и передать ему yScale. Прикрепить к оси метку с текстом «Yearly salary (USD)» (Годовая зарплата (USD))

D.2.2 Добавление осей в скрипичные диаграммы

Листинг D.3 Добавление осей в скрипичные диаграммы (violins.js)

```
const bottomAxis = d3.axisBottom(xScale)
  .tickSizeOuter(0);
innerChart
  .append("g")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxis);
```

Объявить генератор нижней оси и передать ему xScale. Добавить эту ось во внутреннюю диаграмму

```
const leftAxis = d3.axisLeft(yScale);
innerChart
  .append("g")
    .call(leftAxis);
svg
  .append("text")
    .text("Yearly salary (USD)")
    .attr("x", 0)
    .attr("y", 20);
```

Объявить генератор левой оси и передать ему yScale. Добавить эту ось во внутреннюю диаграмму с меткой «Yearly salary (USD)» (Годовая зарплата (USD))

D.2.3 Добавление межквартильных размахов и средних значений в скрипичные диаграммы

Листинг D.4 Межквартильные размахи и средние значения (violins.js)

```
roles.forEach(role => {

  ...

  const width = 8;
  roleContainer
    .append("rect")
      .attr("x", xScale(role.id) - width/2)
      .attr("y", yScale(role.quartiles[2]))
      .attr("width", width)
      .attr("height", yScale(role.quartiles[0]) -
         ➥ yScale(role.quartiles[2]))
      .attr("rx", 4)
      .attr("ry", 4)
      .attr("fill", gray);
```

Добавить элемент прямоугольника для каждой профессии. Прямоугольник центрируется по центральной линии профессии и простирается вертикально от первого до третьего квартиля. Чтобы скруглить углы прямоугольника, нужно настроить атрибуты `rx` и `ry`

```
  roleContainer
    .append("circle")
      .attr("cx", d => xScale(role.id))
      .attr("cy", d => yScale(role.mean))
      .attr("r", 3)
      .attr("fill", white);

};
```

Добавить круг для каждой профессии. Круг центрируется по центральной линии профессии и выравнивается по вертикали с положением среднего значения

D.3 Решения упражнений в главе 7

D.3.1 Создание оси и добавление кругов в диаграмму рассеяния

Листинг D.5 Рисование диаграммы рассеяния (scatterplot.js)

```
const bottomAxisGenerator = d3.axisBottom(xScale);
innerChart
  .append("g")
    .attr("class", "axis-x")
    .attr("transform", `translate(0, ${innerHeight})`)
    .call(bottomAxisGenerator);
```

Объявить генератор нижней оси и добавить ось во внутреннюю диаграмму

```
const leftAxisGenerator = d3.axisLeft(yScale);
innerChart
  .append("g")
    .attr("class", "axis-y")
    .call(leftAxisGenerator);
```

Объявить генератор левой оси и добавить ось во внутреннюю диаграмму

```
svg
  .append("text")
    .text("Estimated population")
    .attr("text-anchor", "end")
    .attr("x", margin.left + innerWidth + 20)
    .attr("y", height - 3)
    .style("font-size", "18px");
svg
  .append("text")
    .text("Max size (m)")
    .attr("dominant-baseline", "hanging")
    .attr("y", 15)
    .style("font-size", "18px");
```

Добавить метку для каждой оси

```
innerChart
  .selectAll(".cetacean")
  .data(data)
  .join("circle")
    .attr("class", "cetacean")
    .attr("cx", d => xScale(
      d.global_population_estimate))
    .attr("cy", d => yScale(d.max_size_m))
    .attr("r", d => rScale(d.max_weight_t))
    .attr("fill", d => colorScale(d.status))
    .attr('fill-opacity', 0.6)
    .attr("stroke", d => colorScale(d.status))
    .attr("stroke-width", 2);
```

Использовать шаблон привязки данных, чтобы добавить один круг для каждого семейства китовых в наборе данных

Установить атрибуты `cx`, `cy`, `r` и `fill` кругов, используя логарифмическую, линейную, радиальную и цветовую шкалы, объявленные ранее. Установить непрозрачность атрибута `fill` равной 60 % и добавить к кругам обводку толщиной 2 пикселя

D.3.2 Создание подсказки

Листинг D.6 Добавление обработчиков событий (scatterplot.js)

```
innerChart
  .selectAll(".cetacean")
  .data(data)
  .join("circle")
    .attr("class", "cetacean")
    ...
    .on("mouseenter", showTooltip)
    .on("mouseleave", hideTooltip);
```

Добавить обработчик события входа указателя мыши в границы круга, вызывающий функцию showTooltip (она объявлена в листинге D.7)

Добавить обработчик события выхода указателя мыши за границы круга, вызывающий функцию showTooltip (она объявлена в листинге D.7)

Листинг D.7 Создание и заполнение всплывающей подсказки (interactions.js)

```
const appendTooltip = () => {
  const tooltip = innerChart
    .append("text")
      .attr("class", "tooltip")
      .attr("text-anchor", "middle")
      .attr("fill", "#192e4d")
      .style("opacity", 0);
};
```

Добавить текстовый элемент SVG во внутреннюю диаграмму, который будет служить подсказкой. Дать ему имя класса "tooltip", чтобы упростить его выборку позже. Установить непрозрачность равной 0

```
const showTooltip = (e, d) => {
  const cx = e.target.getAttribute("cx");
  const cy = e.target.getAttribute("cy");
  const r = e.target.getAttribute("r");

  d3.select(".tooltip")
    .attr("x", cx)
    .attr("y", cy - r - 10)
    .text(d.common_name)
    .transition()
    .style("opacity", 1);
};
```

Когда указатель мыши входит в границы круга, получить координаты и радиус круга из его атрибутов. Установить текст подсказки на основе данных, прикрепленных к кругу. Расположить подсказку над кругом и установить ее непрозрачность равной 100 %

```
const hideTooltip = (e, d) => {
  d3.select(".tooltip")
    .attr("y", -500)
    .style("opacity", 0);
};
```

Когда указатель мыши покидает границы круга, переместить подсказку за пределы диаграммы и установить ее непрозрачность равной 0

Листинг D.8 Обработка событий наведения указателя мыши на выбранные круги (interactions.js)

```
const handleClickOnFilter = (data) => {

  ...

  d3.selectAll(".filter")
    .on("click", (e, datum) => {
      if (!datum.isActive) {

        innerChart
          .selectAll("circle")
          .data(updatedData, d => d.uid)
          .join(
            enter => enter
              .append("circle")
              ...
              .on("mouseenter", showTooltip)
              .on("mouseleave", hideTooltip)
              ...,
            update => update,
            exit => exit
              ...
          )

      }
    });
};
```

Добавить обработчик события наведения указателя мыши на круг, который вызывает функцию showTooltip (она объявлена в листинге D.7)

Добавить обработчик события выхода указателя мыши за пределы круга, который вызывает функцию showTooltip (она объявлена в листинге D.7)

D.4 Решения упражнений в главе 8

D.4.1 Столбчатая диаграмма

Листинг D.9 Передача данных и цветовой шкалы (Chart.js)

```
const Charts = props => {
  ...

  return (
    <h1>Front-end Frameworks</h1>
    <div className='row'>
      ...
      <div className='row'>
        ...
        <div className='col-12'>
          <BarChart
            data={props.data.experience}
            margin={margin}
```

```
            colorScale={colorScale}
          />
        </div>
      </div>
    </div>
  );
};
```

Листинг D.10 Настройка атрибутов прямоугольника (Rectangle.js)

```
const Rectangle = props => {
  return (
    <rect
      x={props.x}
      y={props.y}
      width={props.width}
      height={props.height}
      fill={props.fill}
    />
  )
};
```

Листинг D.11 Подготовка интервальной оси (Axis.js)

```
const AxisBandBottom = props => {
  return (
    <g className="axis" transform={`translate(0, ${props.innerHeight})`} >
      <line x1={0} y1={0} x2={props.innerWidth} y2={0} />
      {props.ticks.map(tick => (
        <text
          key={tick}
          textAnchor="end"
          alignmentBaseline="middle"
          transform={`translate(${props.scale(tick) + props.scale.bandwidth()
          ➥ / 2}, 8) rotate(-90)`}
        >
          {tick}
        </text>
      ))}
    </g>
  );
};
```

Листинг D.12. Рисование столбчатой диаграммы (BarChart.js)

```
import * as d3 from "d3";
import Card from '../UI/Card';
import ChartContainer from '../ChartComponents/ChartContainer';
import Axis from "../ChartComponents/Axis";
import Rectangle from "../ChartComponents/Rectangle";
```

```
const BarChart = props => {
  const width = 300;
  const height = 245;
  const marginBottom = 85;
  const innerWidth = width - props.margin.left - props.margin.right;
  const innerHeight = height - props.margin.top - marginBottom;

  const awarenessData = [];
  props.data.forEach(d => {
    const awareness = {
      id: d.id,
      name: d.name,
      awareness_percentage: d.awareness[d.awareness.length- 1]
        .percentage_question
    };
    awarenessData.push(awareness);
  });
  awarenessData.sort((a, b) => b.awareness_percentage -
    a.awareness_percentage);

  const xScale = d3.scaleBand()
    .domain(awarenessData.map(d => d.name))
    .range([0, innerWidth])
    .padding(0.2);
  const yScale = d3.scaleLinear()
    .domain([0, 100])
    .range([innerHeight, 0]);

  return (
    <Card>
      <h2>Awareness</h2>
      <ChartContainer ... >
        <Axis
          type="band"
          scale={xScale}
          ticks={awarenessData.map(d => d.name)}
          innerWidth={innerWidth}
          innerHeight={innerHeight}
        />
        <Axis
          type="left"
          scale={yScale}
          innerWidth={innerWidth}
          innerHeight={innerHeight}
          label={"Awareness %"}
        />
        {awarenessData.map(framework => (
          <Rectangle
            key={`rectangle-${framework.id}`}
            x={xScale(framework.name)}
            y={yScale(framework.awareness_percentage)}
```

Извлечь данные об осведомленности и отсортировать их в порядке убывания

Объявить интервальную шкалу, отвечающую за распределение прямоугольников по горизонтали, и линейную шкалу для расчета их высоты

Добавить интервальную ось. Передать xScale и отметки

Добавить левую ось. Передать yScale и метку «Awareness %» (Осведомленность %)

Добавить прямоугольник для каждого фреймворка. Передать атрибуты `x`, `y`, `width`, `height` и `fill` как свойства

```
            width={xScale.bandwidth()}
            height={innerHeight - yScale(framework
              .awareness_percentage)}
            fill={props.colorScale(framework.id)}
          />
        ))}
      </ChartContainer>
    </Card>
  )
};
```

Добавить прямоугольник для каждого фреймворка. Передать атрибуты `x`, `y`, `width`, `height` и `fill` как свойства

D.4.2 Визуализация рейтингов

Листинг D.13 Создание компонента Badge (Badge.js)

```
import Circle from "../ChartComponents/Circle";

const Badge = props => {
  return (
    <g style={{ transform: `translate(${props.
      translation[0]}px,${props.translation[1]}px)`}}>
      <Circle
        cx={0}
        cy={0}
        r={18}
        fill={"#fff"}
        stroke={props.strokeColor}
        strokeWidth={3}
      />
      <text
        textAnchor="middle"
        alignmentBaseline="middle"
        fill="#374f5e"
        style={{ fontSize: "12px", fontWeight: "bold" }}
      >
        {props.label}
      </text>
    </g>
  );
};

export default Badge;
```

Сместить элемент группы, используя свойство, которое будет передано из компонента Rankings

Вызвать компонент Circle и передать ему атрибуты как свойства

Отобразить текстовый элемент, получающий свое значение как свойство

Листинг D.14 Вызов компонента Badge (Rankings.js)

```
const Rankings = props => {
  ...

  return (
    <Card>
      <h2>Rankings</h2>
```

```
      <RankingFilters ... />
      <ChartContainer ... >
        ...
        {props.data.experience.map((framework, i) => (
          <g key={`curve-${framework.id}`}>
            <Curve ... />
            ...
            {framework[activeFilter].map((selection,i)
              => (                                    <Fragment
                key={`${framework.id}-selection-${i}`}>
              {selection.rank &&
                <Badge
                  translation={[xScale(selection
                  .year), yScale(selection.rank)]}
                  strokeColor={props.colorScale(framework.id)}
                  label={`${Math.round(selection.percentage_question)}%`}
              />
                }
              </Fragment>
            ))}
          </g>
        ))}
      </ChartContainer>
    </Card>
  )
};
```

Добавить обертку Fragment для каждого элемента в массиве, соответствующем активному фильтру, например массиву удовлетворенности

Если свойство rank не равно null, то отобразить компонент Badge. Передать его смещение, цвет обводки и метку как свойства

D.5 Решения упражнений в главе 9

D.5.1 Изменение ориентации столбчатой диаграммы на экране мобильного устройства

Листинг D.15 Создание горизонтальной столбчатой диаграммы (BarChartMobile.js)

```
import * as d3 from "d3";
import Card from "../UI/Card";
import ChartContainer from "../ChartComponents/ChartContainer";
import Axis from "../ChartComponents/Axis";
import Rectangle from "../ChartComponents/Rectangle";

const BarChartMobile = props => {
  const width = 300;
  const height = 400;
  const marginRight = 38;
```

```
const marginLeft = 95;
const innerWidth = width - marginLeft - marginRight;
const innerHeight = height - props.margin.top - props.margin.bottom;

const awarenessData = [];
props.data.forEach(d => {
  const awareness = {
    id: d.id,
    name: d.name,
    awareness_percentage: d.awareness[d.awareness.length -
    ➥ 1].percentage_question
  };
  awarenessData.push(awareness);
});
awarenessData.sort((a, b) => b.awareness_percentage -
➥ a.awareness_percentage);

const xScale = d3.scaleLinear()
  .domain([0, 100])
  .range([0, innerWidth]);
const yScale = d3.scaleBand()
  .domain(awarenessData.map(d => d.name))
  .range([0, innerHeight])
  .padding(0.2);

return (
  <Card>
    <h2>Awareness</h2>
    <ChartContainer
      width={width}
      height={height}
      margin={{ top: props.margin.top, right: marginRight, bottom:
      ➥ props.margin.bottom, left: marginLeft }}
    >
      <Axis
        type="bandLeft"
        scale={yScale}
        ticks={awarenessData.map(d => d.name)}
        innerWidth={innerWidth}
        innerHeight={innerHeight}
      />
      {awarenessData.map(framework => (
        <g
          key={`rectangle-${framework.id}`}
          className="axis"
        >
          <Rectangle
            x={0}
            y={yScale(framework.name)}
```

```
              width={xScale(framework.awareness_percentage)}
              height={yScale.bandwidth()}
              fill={props.colorScale(framework.id)}
            />
            <text
              x={xScale(framework.awareness_percentage) + 5}
              y={yScale(framework.name) + yScale.bandwidth()/2}
              alignmentBaseline="middle"
              style={{ fontSize: "13px" }}
            >
              {`${Math.round(framework.awareness_percentage)}%`}
            </text>
          </g>
        ))}
      </ChartContainer>
    </Card>
  );
};

export default BarChartMobile;
```

D.6 Решения упражнений в главе 10

D.6.1 Создание узоров SVG

Листинг D.16 Создание узора из вертикальных полос (scatterplot.js)

```
const verticalLinesPattern = defs
  .append("pattern")
    .attr("id", "pattern-vertical-lines")
    .attr("width", 4)
    .attr("height", 4)
    .attr("patternUnits", "userSpaceOnUse");
verticalLinesPattern
  .append("rect")
    .attr("x", 0)
    .attr("y", 0)
    .attr("width", "100%")
    .attr("height", "100%")
    .attr("fill", "#fff");
verticalLinesPattern
  .append("line")
    .attr("x1", 3)
    .attr("y1", 0)
    .attr("x2", 3)
    .attr("y2", 4)
    .attr("stroke", colorScale("VU"));
```

Добавить новый узор в элемент defs и задать его идентификатор равным "pattern-vertical-lines". Определить ширину и высоту равными 4 пикселям и присвоить его атрибуту `patternUnits` значение userSpaceOnUse, чтобы гарантировать соответствие размеров его потомков системе координат контейнера SVG

Добавить элемент прямоугольника белого цвета, который будет играть роль плитки

Добавить в середину контейнера узора линию. Присвоить ее атрибуту `fill` цвет, связанный с охранным статусом "VU"

D.7 Решения упражнений в главе 11

Листинг D.17 Добавление раздела для древовидной карты (index.html)

```
<!DOCTYPE html>
<html>
  ...
<body>
  <div class="container">

    ...

    <section>
      <h2>Treemap</h2>
      <div id="treemap"></div>
    </section>
  </div>

  <script type="module" src="./js/main.js"></script>
</body>
</html>
```

Добавить раздел, содержащий div с идентификатором "treemap"

Листинг D.18 Рисование древовидной карты (treemap.js)

```
import { treemap } from "d3-hierarchy";
import { select, selectAll } from "d3-selection";
import { colorScale } from "./scales";

export const drawTreemap = (root, leaves) => {
  const width = 850;
  const height = 600;

  const treemapLayoutGenerator = treemap()
    .size([width, height]))
    .paddingInner(1)
    .paddingOuter(1)
    .round(true);
  treemapLayoutGenerator(root);

  const svg = select("#treemap")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`);

  const nodes = svg
    .selectAll(".node-container")
    .data(leaves)
    .join("g")
      .attr("class", "node-container")
      .attr("transform", d => `translate(${d.x0},
        ${d.y0})`);
```

Создать генератор макета древовидной карты. Задать размеры диаграммы и отступы между прямоугольниками. Передать иерархические данные (корень) в генератор макета

Добавить контейнер SVG в div с идентификатором "treemap"

Используя шаблон привязки данных, добавить группу SVG для каждого листового узла и переместить группу в местоположение, указанное ключами `x0` и `y0` в данных, вычисленных генератором макета. Этот шаг упростит совместное размещение прямоугольников и меток

```
  nodes
    .append("rect")
      .attr("class", "treemap-node")
      .attr("x", 0)
      .attr("y", 0)
      .attr("width", d => d.x1 - d.x0)
      .attr("height", d => d.y1 - d.y0)
      .attr("rx", 3)
      .attr("ry", 3)
      .attr("fill", d => colorScale(
        d.parent.data.parent));
```

Добавить прямоугольник для каждого листового узла. Их ширину можно вычислить по разнице между `x1` и `x0`, а высоту – по разнице между `y1` и `y0`. Настроить их атрибут `fill` с помощью цветовой шкалы на основе семейства языков

```
  nodes
    .append("text")
      .attr("class", d => `treemap-label day
        -${d.id.replaceAll(" ", "-")
        .replaceAll("'", "")}`)
      .attr("x", 5)
      .attr("y", 15)
      .attr("fill", "white")
      .style("font-size", "12px")
      .style("font-weight", 500)
      .text(d => d.id);
```

Добавить текстовый элемент для каждого листового узла. Расположить его над соответствующим прямоугольником и добавить текст с названием языка

```
  selectAll(".treemap-label")
    .style("opacity", d => {
      const textElement = document.querySelector(
        `.treemap-label-${d.id.replaceAll(" ", "-")
        .replaceAll("'", "")}`);
      const textWidth = textElement.getBBox().width;
      return ((d.y1 - d.y0) >= 25) && ((d.x1 - d.x0)
        >= textWidth + 10) ? 1 : 0;
    });
};
```

Необязательно: скрыть текстовые метки, которые меньше соответствующих им прямоугольников

Листинг D.19 Вызов drawTreemap (main.js)

```
...
import { drawTreemap } from "./treemap.js";

...

drawTreemap(root, leaves);
```

D.8 Решения упражнений в главе 12

Листинг D.20 Рисование матрицы смежности (matrix.js)

```
import { select, selectAll } from "d3-selection";
import { max } from "d3-array";
import { scaleLinear } from "d3-scale";

export const drawMatrix = (nodes, edges) => {

  ...

  const innerWidth = nodes.length * (squareWidth +
    padding);
  const innerHeight = nodes.length * (squareWidth +
    padding);
  const margin = { top: 130, right: 0,
    bottom: 0, left: 130 };
  const width = innerWidth + margin.right +
    margin.left;
  const height = innerHeight + margin.top +
    margin.bottom;
```

Рассчитать ширину и высоту матрицы на основе количества узлов, их размеров и отступов между ними

```
  const svg = select("#matrix")
    .append("svg")
      .attr("viewBox", `0 0 ${width} ${height}`)
    .append("g")
      .attr("transform", `translate(${margin.left},
        ${margin.top})`);
```

Добавить контейнер SVG в div с идентификатором "matrix" и группой для внутренней диаграммы в соответствии со стратегией, описанной в главе 4

```
  const maxWeight = max(edges, d => d.weight);
  const opacityScale = scaleLinear()
    .domain([0, maxWeight])
    .range([0, 1]);

  svg
    .selectAll(".grid-quare")
    .data(matrix)
    .join("rect")
      .attr("class", "grid-quare")
      .attr("x", d => d.x)
      .attr("y", d => d.y)
      .attr("width", squareWidth)
      .attr("height", squareWidth)
      .attr("fill", "#364652")
      .attr("fill-opacity", d =>
        opacityScale(d.weight));

  const labelsContainer = svg
    .selectAll(".matrix-label")
    .data(nodes)
```

Добавить элемент прямоугольника для каждого элемента в массиве матрицы. Установить его местоположение согласно значениям ключей `x` и `y`, а его атрибуту `fill` присвоить цвет «#364652». Вычислить значение для fill-opacity на основе веса ссылки. Максимально доступный вес получает непрозрачность 100 %

```
    .join("g")
      .attr("class", "matrix-label")
      .attr("dominant-baseline", "middle")
      .style("font-size", "13px");

  labelsContainer
    .append("text")
      .attr("class", "label-top")
      .attr("x", -8)
      .attr("y", (d, i) => i * (squareWidth + padding)
        + squareWidth / 2)
      .attr("text-anchor", "end")
      .text(d => d.name);

  labelsContainer
    .append("text")
      .attr("class", "label-left")
      .attr("transform", (d, i) => `translate(
       ${i*(squareWidth+padding) + squareWidth/2}, -8)
       rotate(-90)`)
      .text(d => d.name);

});
```

Добавить метку для каждого узла с левой стороны сетки

Добавить метку для каждого узла в верхней части сетки. Повернуть текстовые элементы вертикально

D.9 Решения упражнений в главе 13

Листинг D.21 Города, где родились лауреаты Нобелевской премии (map-france.js)

```
...
import { max } from "d3-array";
import { getCityRadius } from "./scales";

export const drawFranceMap = (laureates, france) => {

  ...

  const franceLaureates = laureates.filter(laureate =>
    laureate.birth_country === "France");

  const cities = [];
  franceLaureates.forEach(laureate => {
    if (cities.find(city => city.name ===
      laureate.birth_city)) {cities.find(city =>
      city.name === laureate.birth_city)
      .laureates.push(laureate);
    } else {
      cities.push({
        name: laureate.birth_city,
        latitude: laureate.birt_city_latitude,
```

Отфильтровать данные о лауреатах, чтобы оставить только тех, кто родился во Франции

Создать массив с городами во Франции, в которых родились лауреаты. Для каждого города сохранить его широту, долготу и массив со списком лауреатов

```
        longitude: laureate.birt_city_longitude,
        laureates: [laureate]
      });
    }
  });
  const maxLaureatesPerCity = max(cities, d => d.laureates.length);

  svg
    .selectAll(".france-city-circle")
    .data(cities)
    .join("circle")
      .attr("class", "france-city-circle")
      .attr("cx", d => projection([d.longitude,
        d.latitude])[0])
      .attr("cy", d => projection([d.longitude,
        d.latitude])[1])
      .attr("r", d => getCityRadius(d.laureates.length,
        maxLaureatesPerCity))
      .attr("fill", "#35a7c2")
      .attr("fill-opacity", 0.5)
      .attr("stroke", "#35a7c2");

};
```

Создать массив с городами во Франции, в которых родились лауреаты. Для каждого города сохранить его широту, долготу и массив со списком лауреатов

Использовать шаблон привязки данных, чтобы добавить круг для каждого города

Передать долготу и широту города в проекцию, чтобы получить его местоположение на карте

Вызвать getCityRadius, чтобы получить радиус круга

D.10 Решения упражнений в главе 14

Листинг D.22 Импорт набора данных писем (Grid.svelte)

```
<script>
  import letters from "../data/letters.json";

  ...
</script>

<svelte:window bind:innerWidth={windowWidth} />

{#if svgWidth && svgHeight}
  <svg width={svgWidth} height={svgHeight}>
    {#each years as year, i}
      <g
        transform="translate(
           {(i % numColumns) * smWidth},
           {Math.floor(i / numColumns) * smHeight})"
      >
        <rect x={0} y={0} width={smWidth} height={smHeight} />
        <GridItem
          {smWidth}
          {smHeight}
          {year}
          {paintingAreaScale}
```

```
          {paintingDefaultRadius}
          paintings={paintings.filter((painting) =>
            painting.year === year)}
          {maxDrawings}
          drawings={yearlyDrawings.find((d) => d.year === year).months}
          letters={letters.filter((letter) =>
            letter.year === year)}
        />
      </g>
    {/each}
  </svg>
{/if}
```

Передать письма, соответствующие текущему году, в компонент GridItem

Листинг D.23 Передача данных в Letters.svelte (GridItem.svelte)

```
<script>
  import Letters from "../chart_components/Letters.svelte";

  export let letters;

  ...
</script>

<g transform="translate({smWidth / 2}, 0)">
  <g transform="translate(0, {padding + radius})">
    <circle ... />
    ...
    <Drawings {drawings} {monthScale} {radialScale} />
    <Letters {letters} {monthScale} {radialScale} />
  </g>
  <text ... >{year}</text>
</g>
```

Передать необходимые свойства компоненту Letters

Листинг D.24 Рисование радиальной столбчатой диаграммы (Letters.svelte)

```
<script>
  import { months } from "../utils/months";

  export let letters;
  export let monthScale;
  export let radialScale;
</script>

{#each months as month}
  <line
   x1={0}
   y1={0}
   x2={radialScale(letters.find(l => l.month===month)
     .number_of_letters)
     * Math.sin(monthScale(month))}
   y2={radialScale(letters.find(l => l.month===month)
     .number_of_letters)
     * Math.cos(monthScale(month))}
```

Добавить элемент линии для каждого месяца

Установить точку начала линии в центр координат

Рассчитать координаты конечной точки, передав количество писем радиальной шкале и используя тригонометрическую функцию

```
    stroke-width={2}
    stroke-linecap="round"
  />
{/each}

<style lang="scss">
  line {
    stroke: $secondary;
    pointer-events: none;
  }
</style>
```

Листинг D.25 Рисование временной шкалы (Timeline.svelte)

```
<script>
  import { scaleTime } from "d3-scale";
  import { timeFormat } from "d3-time-format";
  import timeline from "../data/timeline.json";

  let height;

  const startDate = new Date(
    timeline[0].start_year,
    timeline[0].start_month,
    1
  );
  const endDate = new Date(
    timeline[timeline.length - 1].end_year,
    timeline[timeline.length - 1].end_month,
    0
  );

  $: timeScale = scaleTime()
       .domain([startDate, endDate])
       .range([0, height]);
</script>

<div class="timeline-container"
  bind:clientHeight={height}>
  {#each timeline as period}
    <div
      class="period-container"
      style="top: {timeScale(new Date(
                      period.start_year,
                      period.start_month,
                      1))}px;
             height: {timeScale(new Date(
                        period.end_year,
                        period.end_month,
                        25)) -
                      timeScale(new Date(
                        period.start_year,
                        period.start_month,
                        0))}px;"
    >
```

Установить начальную дату временной шкалы, создав новый JavaScript-объект Date

Аналогично поступить с конечной датой, передав 0 в качестве дня, чтобы указать последний день месяца

Объявить временную шкалу, которая принимает дату и возвращает соответствующее вертикальное положение на временной шкале

Связать значение переменной height с фактической высотой контейнера временной шкалы

Выполнить цикл по каждому периоду в наборе данных временной шкалы

Настроить стилевые свойства top и height каждого периода, вызвав временную шкалу и передав ей даты периодов

```
    <div class="dates-container">
      <div class="start-date">
        {timeFormat("%b %Y")(new Date(
                              period.start_year,
                              period.start_month,
                              1))}
      </div>
      <div class="end-date">
        {timeFormat("%b %Y")(new Date(
                              period.end_year,
                              period.end_month,
                              1))}
      </div>
    </div>
    <div class="period" />
      <div class="location">
        <div class="city">{period.city}</div>
        <div class="country">({period.country})</div>
      </div>
    </div>
  {/each}
</div>

<style lang="scss">
  .timeline-container {
    height: 100%;
    font-size: 1.6rem;
    line-height: 1.2;
    color: $secondary;
  }
  .period-container {
    display: flex;
    align-items: stretch;
    position: absolute;
    left: 0;
    padding: 3px 0;
    cursor: pointer;
    transition: opacity 100ms ease;
  }
  .period {
    width: 10px;
    background-color: rgba($secondary, 0.7);
    border-radius: 8px;
  }
  .dates-container {
    display: flex;
    flex-direction: column;
    justify-content: space-between;
    width: 70px;
  }
```

```
  .location {
    display: flex;
    flex-direction: column;
    justify-content: center;
    width: 120px;
    padding-left: 10px;
  }
  .city {
    font-weight: 600;
  }
</style>
```

D.11 Решения упражнений в главе 15

Листинг D.26 Стилизация контейнера холста (index.html)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Gallery of Canvas Shapes | D3.js in Action</title>
</head>
<body>
  <div id="canvas" style="max-width: 400px;
    margin: 0 auto;"></div>

  <script src="https://d3js.org/d3.v7.min.js"></script>
  <script src="./js/main.js"></script>
</body>
</html>
```

Гарантировать, что div, содержащий элемент canvas, не превысит в ширину 400 пикселей и будет расположен в центре экрана

Листинг D.27 Добавление холста и рисование фигур (main.js)

```
const canvas = d3.select("#canvas")
  .append("canvas")
    .style("border", "1px solid gray")
    .style("max-width", "100%");
```

Добавить элемент Canvas в DOM, придать ему серую границу и гарантировать, что он никогда не станет шире своего контейнера

```
const container = document.querySelector("#canvas");
const devicePixelRatio = window.devicePixelRatio;

const width = container.offsetWidth;
const height = width;

canvas
  .attr("width", width * devicePixelRatio)
  .attr("height", height * devicePixelRatio);
```

Настроить атрибуты `width` и `height` элемента Canvas и скорректировать их с учетом плотности пикселей экрана

Инициализировать двумерный контекст и масштабировать его с учетом плотности пикселей экрана

```
const context = canvas.node().getContext("2d");
context.scale(devicePixelRatio, devicePixelRatio);
```

```
context.rect(100, 100, 200, 200);
context.lineWidth = 3;
context.stroke();
```

Нарисовать в центре Canvas квадрат 200 на 200 пикселей с черной обводкой шириной 3 пикселя

```
context.beginPath();
context.arc(200, 200, 100, 0, 2 * Math.PI);
context.fillStyle = "plum";
context.fill();
```

Нарисовать круг с радиусом 200 пикселей и цветом заливки "plum"

```
context.beginPath();
context.moveTo(100, 100);
context.lineTo(300, 300);
context.moveTo(300, 100);
context.lineTo(100, 300);
context.stroke();
```

Нарисовать в квадрате две черные диагональные линии шириной 3 пикселя

```
context.font = "18px sans-serif";
context.fillStyle = "#000";
context.textAlign = "center";
context.fillText("Canvas is awesome!", 200, 90);
```

Добавить метку над квадратом, центрированную внутри элемента Canvas

Приложение E
Очень краткое введение в Svelte

В этом приложении рассматриваются:

- структура файла Svelte;
- обмен информацией между компонентами;
- добавление логики отображения;
- использование реактивных переменных и функций.

В данном приложении представлен обзор Svelte – библиотеки JavaScript, которая широко используется в сочетании с D3.js. Если вы незнакомы с Svelte, то это приложение даст вам знания, необходимые для работы над проектами в главах 14 и 15.

E.1 Структура файла Svelte

Файлы Svelte можно узнать по расширению `.svelte`. По соглашению их имена начинаются с заглавной буквы. Файлы Svelte состоят из трех разделов: сценария, разметки и определения стилей. В начале файла находятся теги `<script>` с кодом на JavaScript, реализующем поведение компонента. Затем следует разметка, как, например, тег `<div>` в следующем фрагменте. Обратите внимание, что в разметке можно динамически использовать переменные JavaScript, заключая их в фигурные скобки ({}). И в конце файла находятся теги `<style>`, определяющие стили CSS для оформления компонента. Обратите внимание, что сценарии и стили применяются только к компоненту, в котором они объявлены, и не будут влиять на другие компоненты. Кроме того, компонент не обязательно должен включать все три раздела – они все необязательны. Компонент может содержать только статическую разметку и стили или только логику:

```
<script>
  const myFavFramework: "Svelte";
  const myVizLibrary: "D3.js";
</script>

<div>Я предпочитаю использовать {myVizLibrary} + {myFavFramework}!</div>

<style>
  h1 {
    color: magenta;
  }
</style>
```

При желании можно использовать синтаксис Sass (SCSS) поверх обычного CSS, как в нашем проекте, для этого достаточно добавить атрибут языка в тег `<style>`:

```
<style lang="scss">
  // Здесь находятся определения стилей.
</style>
```

E.2 Передача свойств из родительского компонента в дочерний

Для передачи переменных из родительского компонента в дочерний используются *свойства* (или *реквизиты*). Допустим, следующий фрагмент – родительский компонент `Parent.svelte`. В теге `<script>` мы импортируем дочерний компонент и объявляем константу с именем `myVizLibrary`. В разметке вызываем дочерний компонент с помощью `<Child />` и передаем ему `myVizLibrary` в свойстве с именем `library`:

```
<script>
  import Child from "./Child.svelte";

  const myVizLibrary: "D3.js";
</script>

<div>My new hobby:</div>
<Child library={myVizLibrary} />
```

Затем в дочернем компоненте нужно объявить свойства с помощью ключевого слова `export`. Это может показаться странным, потому что в JavaScript экспорт работает не так, но именно таким способом Svelte принимает свойства в дочерних компонентах.

Следующий фрагмент показывает структуру `Child.svelte`. После объявления свойства `library` его можно свободно использовать в теге `<script>` и в разметке. Каждый раз, когда свойство обновля-

ется в родительском компоненте, новое значение автоматически передается в дочерний компонент. Обратите внимание, что этот прием можно использовать для передачи не только переменных, но и функций:

```
<script>
  export let library;
</script>

<div>Learning {library}!</div>
```

Если имя константы в родительском элементе совпадает с именем свойства в дочернем элементе, мы можем упростить синтаксис `<Child library={myVizLibrary} />` до `<Child {library} />`.

E.3 Передача информации из дочернего компонента в родительский

Иногда в проектах также может потребоваться передать информацию из дочернего компонента в родительский, например при обнаружении события. В следующем фрагменте показан родительский компонент, импортирующий дочерний компонент и объявляющий переменную с именем `isTooltipVisible`. Затем мы не только передаем `isTooltipVisible` дочернему компоненту как свойство, но и используем ключевое слово `bind`:

```
<script>
  import Child from "./Child.svelte";
  import Tooltip from "./Tooltip.svelte";

  let isTooltipVisible = false;
</script>

<Child bind:tooltip={isTooltipVisible} />

{#if isTooltipVisible}
  <Tooltip />
{/if}
```

Дочерний компонент, обнаружив щелчок на кнопке, вызывает функцию `handleClickOnButton()`, которая переключает значение свойства `tooltip`. Поскольку мы использовали ключевое слово `bind`, то соответствующее значение будет обновлено в родительском компоненте:

```
<script>
  export let tooltip;
```

```
  const handleClickOnButton = () => {
    tooltip = !tooltip;
  };
</script>

<button on:click={handleClickOnButton}>Show tooltip</button>
```

E.4 Добавление логики отображения

Добавление логики отображения в разметку позволяет легко контролировать, какая часть дочерних компонентов должна отображаться в зависимости от взаимодействий с пользователем или других динамических факторов. В следующем примере мы вновь используем предыдущий родительский компонент, в котором блок {#if} отображает компонент подсказки всякий раз, когда `isTooltipVisible` получает значение `true`:

```
<script>
  import Child from "./Child.svelte";
  import Tooltip from "./Tooltip.svelte";

  let isTooltipVisible = false;
</script>

<Child bind:tooltip={isTooltipVisible} />

{#if isTooltipVisible}
  <Tooltip />
{/if}
```

Еще один логический блок, блок {#each}, можно использовать для отображения каждого элемента в списке. Этот блок работает подобно циклу `for`:

```
<script>
  const myListItems = ["item A", "item B", "item C"];
</script>

<ul>
  {#each myListItems as item}
    <li>{item}</li>
  {/each}
</ul>
```

Этот компонент отобразит следующий неупорядоченный список:

```
// => <ul>
        <li>item A</li>
        <li>item B</li>
        <li>item C</li>
      </ul>
```

E.5 Использование реактивных переменных и функций

С помощью ключевого слова `$:` можно приказать Svelte автоматически обновить связанные переменные и функции при присваивании нового значения существующей переменной. В следующем фрагменте мы объявляем переменную с именем `width` и шкалу D3, диапазон которой пропорционален значению `width`. В разметке мы привязываем переменную `width` к значению `innerWidth` объекта `window`. Каждый раз, когда изменяется ширина окна, переменная `width` обновляется соответствующим образом. А поскольку мы добавили ключевое слово `$:` перед объявлением шкалы, то и шкала будет обновляться при каждом изменении переменной `width`:

```
<script>
  import { scaleLinear } from "d3-scale";
  let width;

  $: const myScale = scaleLinear()
    .domain([0, 100])
    .range([0, width]);
</script>

<svelte:window bind:innerWidth={width} />
```

В этом приложении мы представили краткий обзор основных концепций Svelte, которые вам пригодятся при создании своих проектов визуализации данных. Желающие глубже изучить Svelte могут найти краткие и емкие руководства по адресу https://svelte.dev/tutorial.

Предметный указатель

Символы

+ оператор, 62, 112
<body> элемент, 40, 43
<canvas> элемент, 631
<circle /> элемент, 50
<ellipse /> элемент, 50
<g> элемент, 53
 элемент, 231
<path> элемент, 50
<rect /> элемент, 47
<script> тег, 631
 элемент, 59, 231
<svg> контейнер, 31
<text> элемент, 52
$: ключевое слово, 709
2-мерная система координат, 44
*/ комментарий, 138, 272
/* комментарий, 138, 272
// комментарий, 138, 272
{#each} блок (Svelte), 708
.enter().append() методы, 119
-g команда, 456
{#if} блок (Svelte), 708
@types/d3 пакет, 349

A

ActionScript3, 35
activeFilter переменная состояния, 367
addEventListener() метод, 284
addNodeColor() функция, 652
Adobe Color веб-сайт, 590
alignment-baseline атрибут, 296
alphaDecay скорость остывания, 610
angle() функция, 615
Angular фреймворк пользовательского интерфейса, 358
append() метод, 84
Apple, 35
arc() метод, 637
aria-labelledby атрибут, 432, 434
aria-pressed атрибут, 429, 435
attr() метод, 87, 121, 286
attr префикс, 349
Axis компонент, 354, 359, 364
axis() функция, 158
axisBottom() функция, 160
axis.tickFormat() метод, 162

B

background-color атрибут, 232
bandwidth() метод, 135
bandwidth свойство, 216
bandwidth() свойство, 219
base.css файл, 444
beginPath() метод, 634, 638
Ben & Jerry's проект, 66
bind ключевое слово, 707
bind:this атрибут, 349
blur событие, 442
border свойство, 42, 92
border-radius свойство, 183
breakpoint константа, 393

brushHandler константа, 559

C

call() метод, 160, 217, 221, 317
Canvas, 57, 629
  обзор, 629
  отображение
    визуализаций, 628
    кривых, 639
    кругов, 637
    линий, 634
    простых фигур, 630
    прямоугольников, 635
    текста, 639
  смешанный режим отображения, 641
  стратегии взаимодействия с, 649
caption тег, 436
centroid() метод, 186
cetaceanFilters массив, 314
Chartability, 407
children свойство, 339
circle атрибут, 298
clamp() свойство, 380, 414
class атрибут, 121, 124
classed() метод, 286, 287
clientX свойство, 285
clientY свойство, 285
col-12 класс, 392
col-lg-3 класс, 392
col-lg-9 класс, 392
col-lg-12 класс, 392
col-md-6 класс, 392
color свойство, 46, 301
Color Hunt веб-сайт, 590
color-legend класс, 231
colorScale, 309, 353
colorScale константа, 206
columns атрибут, 201
conservationStatuses массив, 310
context.stroke() метод, 635
Coolors веб-сайт, 590
cornerRadius() функция, 183
createTooltip() функция, 294, 301
CSS (Cascading Style Sheets), 57
current свойство, 343
Curve компонент, 360
curve() функция, 170, 172, 220
curveBumpX() функция, 482
cx атрибут, 50, 166, 298, 352, 469, 611
cy атрибут, 50, 166, 298, 317, 352, 469, 611

D

d3, библиотека, установка в проект React, 333
d3 пакет, 334
d3.arc() метод, 205
d3.arc() функция, 183
d3.area() функция, 173
d3-array модуль, 115, 217, 242, 255
d3.autoType метод, 153
d3.autoType функция, 180
d3-axis модуль, 159, 217
d3.axisBottom() метод, 217, 249, 259
d3.axisLeft() генератор осей, 221
d3.axisLeft() метод, 249, 259
d3.axisLeft() функция, 164
d3.bin() метод, 242, 244, 266
d3.brushX() метод, 559
d3.csv() метод, 153, 242, 458
d3.csv() функция, 104, 110, 117, 180
d3.curveBasis функция, 171
d3.curveBundle функция, 171
d3.curveCatmullRom интерполятор, 170
d3.curveCatmullRom метод, 270
d3.curveCatmullRom функция, 172, 220
d3.curveMonotoneX функция, 172
d3.curveMonotoneX функция интерполяции, 360
d3.curveStep функция, 172
D3 (Data Driven Documents), 84
D3 (Data-Driven Documents)
  как вспомогательная библиотека, 350
  подходы к использованию
  во фреймворках пользовательских интерфейсов, 331
  фреймворки пользовательских интерфейсов, 329
  Angular, 348
  Svelte, 349
d3.dsv() функция, 105
d3.easeCircleIn функция, 291
d3.easeCircleInOut функция, 291
d3.easeCircleOut функция, 291
d3.easeCubicInOut функция, 291
d3.easeCubicOut функция, 292

d3.easeExpIn функция, 291
d3.easeExpInOut функция, 291
d3.easeExpOut функция, 291, 319
d3.easeQuadIn функция, 291
d3.easeQuadInOut функция, 291
d3.easeQuadOut функция, 291
d3.extent() метод, 255
d3.extent() модуль, 115
d3-fetch модуль, 109
d3.forceCenter() метод, 515
d3.forceCollide() метод, 514
d3.forceLink() метод, 518
d3.forceManyBody() метод, 516
d3.forceRadial() метод, 512
d3.forceSimulation() метод, 509
d3.forceX() метод, 511
d3.forceY() метод, 511
d3.format(".0%"), 186, 207
d3.format() метод, 298
d3.geoEqualEarth() метод, 535
d3.geoGraticule() функция, 540
d3.geoMercator() метод, 565
d3.geoOrthographic() функция, 540
d3.geoPath() метод, 536
d3.hierarchy() метод, 463
d3.interpolateYlGnBu цветовая схема, 542
D3.js (Data-Driven Documents), 27
  как работает, 31
  когда использовать, 29
  обзор, 27
  экосистема, 36
  HTML и DOM, 37
d3.json() метод, 336
d3.json() функция, 105
d3.line() метод, 335
d3.line() функция, 168, 170
d3.link() функция, 481
d3.max() метод, 115, 158, 215, 266, 484
d3.max() функция, 229, 258, 309
d3.mean() метод, 266
d3.median() метод, 255
d3.min() метод, 115, 158, 266
d3.min() функция, 229, 301
d3.pack() функция, 488
d3.partition() функция, 488
d3.pie() метод, 202, 209
d3.pointer() метод, 305
d3.randomNormal() метод, 514
d3.range() метод, 217
d3-scale модуль, 471, 665
d3.scaleBand() функция, 130, 134, 199
d3.scaleLinear() метод, 335
d3.scaleLinear() функция, 127, 130, 131
d3.scaleLog() метод, 310
d3.scaleOrdinal() функция, 206
d3.scalePoint() метод, 257, 267
d3.scaleQuantile() метод, 255
d3.scaleSequential() метод, 543
d3.scaleSqrt() функция, 313
d3.scaleTime() функция, 158
d3.schemeTableau10 палитра, 344
d3.select() метод, 80, 82
d3.selectAll() метод, 80
d3-selection модуль, 84, 284, 468
d3-shape модуль, 166, 183
d3.stack() метод, 212
d3.stackOffsetDiverging, 227
d3.stackOffsetExpand, 227, 230
d3.stackOffsetNone, 227
d3.stackOffsetSilhouette, 227, 229
d3.stackOffsetWiggle, 227
d3.stackOrderAppearance, 227
d3.stackOrderAscending, 227
d3.stackOrderDescending, 227, 230
d3.stackOrderInsideOut, 227, 229
d3.stackOrderNone, 226
d3.stackOrderReverse, 227
d3.stratify() метод, 459, 464
d3.timeFormat() метод, 162
d3-timeormat модуль, 162
d3.transition() метод, 319
d3.tree() метод, 480
d3.tree() функция, 488
d3.treemap() функция, 488
d3.Treemap() функция, 488
d3.tsv() функция, 105
d3.zoom() метод, 554
d3.zoomIdentity функция, 557
d атрибут, 50, 169, 271, 615
  использование генератора линий, 168
  рисование
    дуг, 205
    линейной диаграммы, 166
  создание кривых, 360
d параметр, 112, 121, 285
data() метод, 118, 315
data переменная состояния, 336

data.columns атрибут, 201
data.columns ключ, 212
Data Visualization Society, 35
Dataviz Accessibility Resources, 446
Date() конструктор, 161, 163
datum() метод, 540
defined() функция, 360
defineScales() функция, 197
defs элемент, 419
delay() метод, 289
dependencies массив, 345
desc тег, 432
descendants() метод, 460, 463
devicePixelRatio свойство, 633, 645
display свойство, 438
displayCities() функция, 550
displayCountries() функция, 550
div элемент, 474
domain класс, 162
domain() метод, 127, 131
DOM (Document Object Model), 73
  атрибут style, 91
  выбор элементов, 80
  добавление элементов в выборку, 84
  подготовка среды, 76
    загрузка D3 в проект, 78
    структура первого проекта D3, 77
  установка и изменение атрибутов, 87
  HTML, 37
DOM (Document Object Model – объектная модель документа), 31
dominant-baseline атрибут, 52, 177, 186, 207, 223
dominant-baseline свойство, 209
drawDonutCharts() функция, 197
drawLegend() функция, 553
drawLineChart() функция, 166
duration() метод, 289, 319
dx атрибут, 223
dy атрибут, 223

E

e параметр, 285
each блок, 602
ease() метод, 319
ease-in функция, 289
ease-in-out функция, 289
ease-out функция, 289
edgeHash объект, 499
edges массив, 499
ElementRef, 348
ellipse() метод, 637
end папка, 39
endAngle значение, 183
endAngle() функция, 205
enter-update-exit шаблон, 95
extent() функция, 560

F

Feature значение, 532
feature() метод, 564
FeatureCollection значение, 532
features объект, 543
fill атрибут, 47, 51, 90, 93, 125, 167, 169, 206, 208, 216, 249, 261, 296, 352, 421, 469, 640
fill() метод, 636
fill прозрачность, 48
fill свойство, 92
fill fnhb,en, 221
fill-opacity атрибут, 48, 175, 208, 296, 484, 500
fill-opacity свойство, 222
fillStyle свойство, 636, 639, 647
fillText() метод, 639
filter() метод, 62, 254, 266
find() метод, 63
Flash, 35
flex свойство, 392
flex-basis свойство, 391
flexbox свойство, 233
flex-grow свойство, 391
flex-shrink свойство, 391
flex-wrap свойство, 389
focus событие, 442
font свойство, 639
font-family свойство, 54, 141, 143, 163, 218
font-size, 379
font-size свойство, 54, 141, 143, 165, 218, 414
font-weight свойство, 218
fontWeight свойство, 362
forceCollide сила столкновения, 610
forceSimulation() метод, 510, 610
forceX() метод, 515

forceX сила позиционирования, 610
forceY сила позиционирования, 610
forceY() функция, 515
forEach() метод, 163
forEach цикл, 62
forEach() цикл, 200
foreignObject элемент, 474
format() функция, 476

G

GeoJSON, 532
geometry свойство, 532
geoPathGenerator, 541
getCityRadius() функция, 549
getContext("2d"), 646
getImageData() метод, 654
getLayout() функция, 393
getMonth() метод, 163
getRadius() функция, 483
GIS (Geographic Information System – географическая информационная система), 107
Google Fonts веб-сайт, 590
graticuleGenerator константа, 540
graticuleGenerator.outline метод, 541
graticules, 540
grid свойство, 392
gridContainer переменная, 594

H

h1 заголовок, 81
handleBrush() функция, 560
handleClickOnFilter() функция, 313
handleMouseEvents() функция, 297, 305
handleMouseMove() функция, 655
handleSimulationEnd() функция, 647, 652
handleWindowResize() функция, 393
height атрибут, 41, 43, 47, 90, 123, 135, 288, 592, 632, 645
height константа, 157, 341
height свойство, 301, 339
Hemingway приложение, 410
hidden класс, 556
histogram.js файл, 243
HTML (Hypertext Markup Language) и DOM, 37

I

id() функция, 460, 518
import инструкция, 459
includes() метод, 61
index.html файл, 40, 77, 196, 457, 631
index.js файл, 335
innerChart выборка, 160, 166
innerChart константа, 157, 182, 244
innerHeight, 341, 351, 354
innerHeight зависимость, 345
innerHeight константа, 156, 159
innerRadius() функция, 183
innerWidth, 341, 351, 354
innerWidth зависимость, 345
innerWidth константа, 156, 159
innerWidth свойство, 380, 393, 414, 594
interactions.js файл, 442
interpolate метод, 471
interpolate() функция, 471
invert() метод, 306, 561

J

JavaScript, 58
  манипуляция массивами, 60
  объединение вызовов методов в цепочки, 59
join() метод, 118, 315, 316
JSON (JavaScript Object Notation – форма записи объектов JavaScript), 105

K

key свойство, 216
keys() функция, 212

L

leaves() метод, 460, 463
leftAxis генератор, 382
leftAxis константа, 164
legend-container класс, 231
legendItems константа, 231
legend.js файл, 313
length свойство, 60, 114, 182, 215
LESS (Leaner Stylesheets), 77
let переменные, 83
line элемент, 45

lineGenerator константа, 169
line-height свойство, 301
lineRadial() метод, 612, 615
lineTo() метод, 635
link() функция, 482
linkGenerator функция, 482
loadCSVData() функция, 458
load-data.js файл, 263, 309, 313, 436
loaddata.js файл, 242
loading переменная состояния, 335

M

main тег, 435
map() метод, 62, 200, 254, 266, 310, 353, 471
Map() объект, 652
map() функция, 134
map-reset кнопка, 557
margin константа, 156
margin объект, 339, 340, 467
margin-left свойство, 386
margin-right свойство, 386
Math.cos() функция, 603
Math.random() функция, 242
Math.round() метод, 306
Math.sin() функция, 603
matrix массив, 499
max() функция, 484
max-height свойство, 476
max-width свойство, 87, 131, 386, 633
max-width тип, 377
MDN Web Docs, 40, 63, 446
mesh() метод, 565
Meta Brains, 331
min-width свойство, 391
min-width тип, 377
mobileData константа, 396
module атрибут, 457
mounted переменная, 337
mouseenter событие, 443
mouseleave событие, 443
mousemove событие, 654
moveTo() метод, 634

N

nativeElement свойство, 348
Netflix, 35
ngAfterViewInit() обработчик событий жизненного цикла, 348
nice() метод, 215, 229, 246, 258, 267, 309
Node Package Manager (NPM), 63, 334, 456
nodes массив, 498
nodes() функция, 510
Notable, 35
npm install команда, 392, 497, 591, 643
npm install d3-interpolate команда, 471
npm install d3-scale команда, 470
npm install d3-shape команда, 481
npm i topojson-client команда, 564
NPM (Node Package Manager), 76
npm run dev команда, 591, 643
npm run start команда, 392
numberOfTicks константа, 355

O

Observable блокноты, 64
offset() функция, 226
offsetX свойство, 285
offsetY свойство, 285
on() метод, 284, 297, 305, 442, 476
onClick событие, 366
onFilterSelection свойство, 366
onMount() метод, 646
opacity свойство, 476, 557, 649
opacity свойство стиля, 317
order() функция, 226
outerRadius() функция, 183
overflow свойство, 556

P

pack() метод, 467
padAngle() функция, 183
padding() метод, 257, 267
padding() функция, 467
paddingInner() свойство, 135
paddingInner() функция, 217, 488
paddingOuter() функция, 488
pageX свойство, 285
pageY свойство, 285
paint-order атрибут, 485
parcel команда, 456
parcel src/index.html команда, 498
parentId() функция, 460

Path2D() конструктор, 638
path элемент, 162, 175, 305, 368, 505, 615
pathRef константа, 368
pattern элемент, 419
patternTransform атрибут, 422
patternUnits атрибут, 420
pieGenerator константа, 203
populateFilters() функция, 283, 313
populateLegend() функция, 313
position свойство, 645
projection константа, 550
projection() функция, 536
properties объект, 543
properties свойство, 532
props зависимость, 345
Protovis, 35

Q

quantiles() функция, 255
querySelectorAll() метод, 581

R

r атрибут, 50, 317, 352
radius() функция, 517, 615
range() метод, 127, 131
React, 274
  загрузка данных в проект, 335
  установка библиотеки D3
  в проект, 333
react-spring библиотека, 332
rect() метод, 636
rect элемент, 47, 90, 118, 139, 420, 422
ref атрибут, 343, 368
remove() метод, 318
require() метод, 462
require() функция, 459
resize событие, 381
resizeChart() функция, 381, 383, 414
resize.js файл, 380, 383, 414
responsive-svg-container класс, 85, 88
role атрибут, 431, 434
root.links() метод, 482
roundRect() метод, 636
row класс, 392
rScale, 309
runas команда, 457
rx атрибут, 48, 295
ry атрибут, 48, 295

S

SASS (Syntactically Awesome
Stylesheets), 77
scale() метод, 634
scale() функция, 535
scaleExtent() функция, 556
scaleOrdinal, 471
scaleRadial() функция, 483
scatterplot.js файл, 441
script тег, 78
select() функция, 468, 481
selectAll() метод, 118
selectAll() функция, 476
selection массив, 561
setData() функция, 336
setLoading() функция, 336
simulation.restart() метод, 523
simulation.stop() метод, 523
simulation.tick() метод, 523
size() функция, 467, 488
slice() метод, 603
sort(), 359
sort() метод, 115, 266
sortable класс, 581
Space Wars (Бремер), 95
span элемент, 303
src атрибут, 581
src папка, 335, 591
srcset атрибут, 581
stackGenerator константа, 212
Stanford, 35
start папка, 39, 77, 109, 153, 197, 240, 282,
300, 308, 386, 497, 591, 640
startAngle значение, 183
startAngle() функция, 205
State of JavaScript Survey 2021, 338
stratify() функция, 460, 463
strength() функция, 512, 518
stroke атрибут, 46, 47, 169, 261, 352, 360,
469, 484, 565
stroke() метод, 636, 639
stroke свойство, 92, 352
stroke-dasharray атрибут, 301
stroke-opacity атрибут, 48

stroke-opacity свойство, 222
strokeStyle свойство, 647
stroke-width атрибут, 46, 49, 261, 352, 505
strokeWidth атрибут, 360
stroke-width свойство, 352
Studio NAND, 274
style атрибут, 42, 54, 91
style() метод, 92, 141, 143
style() vtnjl, 92
sudo команда, 457
sum() метод, 467
Svelte, 274
Svelte фреймворк, 705
  добавление логики отображения, 708
  использование реактивных переменных и функций, 709
  передача информации из дочернего компонента в родительский, 707
  передача свойств из родительского компонента в дочерний, 706
  структура файлов, 705
Svelte фреймворк пользовательского интерфейса, 358
svg константа, 88
SVG контейнер, 87
svg: префикс, 349
svgHeight переменная, 595
SVG (Scalable Vector Graphics), 38, 76
  геометрические примитивы, 44
  группировка элементов, 53
  контейнеры, 42
  определение, 38
  система координат, 44
SVG (Scalable Vector Graphics – масштабируемая векторная графика), 24
svgWidth атрибут, 593
svgWidth переменная, 595
switch оператор, 594
switch() оператор, 471

T

tabindex атрибут, 441, 442
target свойство, 285
text() метод, 141, 176
textAlign свойство, 639
text-anchor атрибут, 53, 140, 186, 207, 223, 296
text-anchor свойство, 163, 209
textAnchor свойство, 362
then() метод, 113, 336
tick класс, 162
ticks массив, 356
ticks() метод, 356, 359
tickSize() метод, 222
tickSizeOuter() метод, 217, 221, 259
tickValues() метод, 217, 221
title тег, 432
tooltip константа, 295
"tooltip-content", 303
tooltipContent константа, 303
tooltipHeight константа, 295
tooltipText константа, 303
tooltipWidth константа, 295
"tooltip-year", 301, 306
TopoJSON, 533, 564
topojson-client библиотека, 534, 565
transform атрибут, 54, 138, 200, 603
transform объект, 555
transform свойство, 54, 369
transform() функция, 557
transition() метод, 289, 293, 298, 367
transition() функция, 535
translate свойство, 138
translate() функция, 537
translateExtent() функция, 556
tree() функция, 480
tspan элемент, 303, 306
tspan элемент SVG, 223
type атрибут, 429, 457
type свойство, 285, 354, 532

U

update функция, 318
updateCityCircles() функция, 550, 563
updateCountryFills() функция, 550, 563
updatedBins константа, 288
updatedData константа, 288
updateHistogram() функция, 286
updateNetwork() функция, 510, 518
useEffect ловушка, 336, 342, 368, 394
useRef ловушка, 343, 368
useRef() ловушка, 342

V

value атрибут, 515
value() функция, 202, 244
viewbox атрибут, 131
viewBox атрибут, 43, 87, 157, 180, 378, 469, 481
ViewChild декоратор, 348
Visual Studio Code (VS Code), 334, 457
VS Code, 76

W

WebAIM contrast checker приложение, 416
WebGL, 634
WebGL (Web Graphics Library), 57
width атрибут, 41, 43, 47, 90, 122, 133, 216, 592, 632, 645
width константа, 157, 341
width свойство, 301, 339
wikitable класс, 581
willReadFrequently свойство, 651
window объект, 380, 414, 594, 633, 645
windowWidth константа, 380
windowWidth переменная, 594

X

x0 атрибут, 270
x0() функция, 272
x1 атрибут, 45, 178, 270
x1() функция, 272
x2 атрибут, 45, 178
x атрибут, 123, 133, 141, 186, 216, 223, 499
x свойство, 611
x() функция, 168, 172, 360, 516
xScale, 131, 158, 199, 267, 269, 306, 309, 353, 360
xScale() функция, 168

Y

y0() функция, 173, 219
y1 атрибут, 45, 178
y(1) функция, 219
y1() функция, 173
y2 атрибут, 45, 178
y атрибут, 123, 141, 186, 223, 288, 499
y свойство, 611
y() функция, 168, 172, 360, 516
yScale, 134, 159, 267, 269, 288, 309, 353, 360
yScale() функция, 169, 176

Z

z-index свойство, 38
zoom событие, 554
zoomHandler, 554

А

Адаптация плотности информации, 393
Адаптивная информационная панель, 385
Адаптивные визуализации
  адаптация плотности информации, 393
  дополнительные советы, 400
  изменение ориентации диаграммы, 398
  информационная панель, 385
  линейные диаграммы, 378
  минималистский подход, 382
  настройка меток осей, 381
Адаптивные сетки, 385
Адаптивный дизайн, 376
  подход с ориентацией преимущественно на мобильные устройства, 377
  подход с ориентацией преимущественно на настольные компьютеры, 377
Азат Мардан (Azat Mardan), 331
Азимутальные проекции, 535
Альберто Кайро, 66
Анимация появления, обновления и исчезновения элементов визуализации, 307
Атрибуты, 37
  динамическая настройка с помощью данных, 121
  установка и изменение, 87

В

Веб-контент, доступность, 406
Визуализация
  взаимодействия, 441
    возможность осуществлять взаимодействия с помощью клавиатуры, 441
    навигация и взаимодействия должны быть простыми, 443
    подсказки, помогающие осуществлять взаимодействия, 444
    эквивалентный опыт на экранах всех размеров, 446
  геопространственной информации, 530
  доступность, 405
    визуальные эффекты не должны представлять опасности для страдающих эпилептическими припадками, 446
    Руководящие принципы доступности веб-контента (Web Content Accessibility Guidelines, WCAG), 407
  позиционирование нескольких ящичных диаграмм на графике, 256
  фильтрация, 282
    создание плавных переходов, 289
Визуальная информация, 415
  данные должны представляться с соответствующей плотностью, 426
  информация должна передаваться не только с помощью цвета, 418
  контрастность цветов, 416
Вложение, 453, 584
Временная шкала, 668
Всплывающие подсказки, 293
  создание, 294
  составные подсказки, 300
Выбор
  значимых взаимодействий, 619
  шкалы, 665
  элементов DOM, 80
Выборка
  исчезновения, 316
  обновления, 316
  появления, 316
Выделение мышью, 558
Вычисление центра дуги, 185

Г

Генератор
  кругового макета, 200
    форматирование данных, 201
  линий, 168
Географическая информационная система (ГИС), 35, 530
Географические данные, 107, 532
Геометрические примитивы, 44
Гибридный подход, 366
Гистограммы, 65, 245
Голосовые помощники, 407
Графы, 493
Группировка данных, 242

Д

Данные, 100, 102
  адаптация под размеры экрана, 126
    шкалы, 127
  загрузка в проект React, 335
  подготовка, 108
    загрузка данных в проект D3, 109
    оценка набора данных, 113
    форматирование набора данных, 111
  поиск, 102
  привязка к элементам DOM, 117
  типы, 103
  форматы и структуры, 104
    географические, 107
    иерархические, 106
    неструктурированные, 108
    сети, 107
    табличные, 104
    JSON, 105
Дашборды бизнес-аналитики, 400
Двойное кодирование, 418
Деревья, 65
Дея Банкова (Dea Bankova), 534
Джонатан Швабиш, 65
Диаграммы
  Вороного, 570

добавление меток, 140
рассеяния, 309
Диапазон, 127
спектра возможных выходных значений, 665
Дискретные данные, 127, 583
Дискретные значения, 665
Дискретный входной диапазон, дискретный выходной диапазон, 680
Дискретный входной диапазон, непрерывный выходной диапазон, 679
Добавление
меток осей, 165
радиальных осей, 599
Дополнительные советы, адаптивные визуализации, 400
Доступность
визуализации, 405
взаимодействия, 441
визуальные эффекты не должны представлять опасности для страдающих эпилептическими припадками, 446
возможность осуществлять взаимодействия с помощью клавиатуры, 441
навигация и взаимодействия должны быть простыми, 443
подсказки, помогающие осуществлять взаимодействия, 444
эквивалентный опыт на экранах всех размеров, 446
визуальная информация, 415
данные должны представляться с соответствующей плотностью, 426
информация должна передаваться не только с помощью цвета, 418
контрастность цветов, 416
экранные дикторы, 426
доступность визуализированных данных для чтения, 433
контейнеры SVG, 431
пользовательские настройки клавиатуры не должны совпадать с комбинациями управления экранным диктором, 439
предоставление таблиц данных, 435
семантическая разметка, 427
Древовидная диаграмма, 478
рисование, 481
Дуги, 152, 179
вычисление центра дуги, 185
генераторы дуг, 183
полярная система координат, 180
Дуговые диаграммы, 503

З

Загрузка данных в проект React, 335

И

Идентичности шкала, 669
Иерархические визуализации, 453
древовидная диаграмма, 478
рисование, 481
другие, создание, 487
упакованная пузырьковая диаграмма
добавление всплывающих подсказок, 476
добавление меток, 474
макет, 465
рисование, 468
упакованная пузырьковая диаграмма, построение, 465
форматирование иерархических данных, 457
файлы CSV, 458
файлы JSON, 461
Иерархические данные, 106
Изабель Мейреллиш, 66
Изменение ориентации диаграммы, 398
Изучение данных, 583
Инициализация и вызов генератора кругового макета, 203
Интерактивность
лучшие практики, 281
причины использования, 281
Интерактивные визуализации, 280
анимация, 307
всплывающие подсказки, 293
создание, 294
диаграммы рассеяния, 309
радиальная шкала, 311
создание переходов для повторного использования, 319

составные подсказки, 300
Интервальная шкала, 134, 679
Интерполяция точек данных кривой, 170
Информационная панель, адаптивная, 385

К

Картограммы, 542
Картографическая сетка, 539
Карты-анаморфозы, 570
Каскадные таблицы стилей (Cascading Style Sheets, CSS), 57
Категориальные цветовые шкалы, 584
Качественные данные, 583
Квадратного корня шкала, 670
Квантилей шкала, 678
Квантильная шкала, вычисление квартилей, 254
Квантования шкала, 677
Квартили, 254
Кнопки, семантические элементы, 428
Количественные данные, 583
Кольцевые диаграммы
  добавление меток, 207
  подготовительные шаги, 197
  создание, 197
Конические проекции, 535
Контейнеры SVG, 338
Корзины, 242
Кривые, 152
  интерполяция точек данных, 170
  создание, 360
Кристиан Суайнхарт, 66
Кристина Сюч (Krisztina Szűcs), 145
Круговые диаграммы
  добавление меток, 207
  подготовительные шаги, 197
  создание, 197
    инициализация и вызов генератора кругового макета, 203
Круговые макеты, 194

Л

Легенда, добавление в проект, 230
Линдси Поултер, 66
Линейная шкала, 130, 668
Линейные диаграммы, 166
  адаптивные, 378
  генераторы линий, 168
  интерполяция точек данных кривой, 170
Линейные преобразования, 666
Логарифмическая шкала, 310, 671
Логарифмические преобразования, 666
Логические пиксели, 633

М

Майк Бостоk, 67
Максимилиан Шварцмюллер (Maximilian Schwarzmüller), 331
Массивы, манипуляция, 60
Масштабирование и панорамирование, 554
Масштабируемая векторная графика (Scalable Vector Graphics, SVG), 24, 38
  определение, 38
Матрица смежности, 497
Местоположение, 584
Метки, 176
  добавление в диаграммы, 140
Минималистский подход, 382
Многочастичная сила, 516
Мозаичные карты, 569
Мортен Барклунд (Morten Barklund), 331

Н

Наборы данных
  загрузка данных в проект D3, 109
  оценка, 113
  форматирование, 111
Надия Бремер, 67, 94
Настройка меток осей, 381
Неориентированные ребра, 496
Непрерывные данные, 127, 583
Непрерывные значения, 665
Непрерывные цветовые шкалы, 584
Непрерывные шкалы, 667
Непрерывный входной диапазон, дискретный выходной диапазон, 677
Непрерывный входной диапазон, непрерывный выходной диапазон, 666
Нестандартные визуализации, 579

выбор значимых взаимодействий, 619
добавление радиальных осей, 599
изучение данных, 583
применение силовой компоновки к элементам на окружности, 605
радиальные диаграммы с областями, 612
радиальные столбчатые диаграммы, 616
сбор данных, 579
создание адаптивных контейнеров SVG, 592
создание адаптивных сеток SVG, 596
создание основы проекта, 591
создание радиальных визуализаций, 599
эскиз макета, 589
Неструктурированные данные, 108
Неупорядоченные списки, 428
Номинальные данные, 583

О

Области размаха, 173
генератор областей размаха, 173
улучшение читаемости с помощью меток, 176
Обработка событий, 284
Объектная модель документа (Document Object Model, DOM), 31
HTML, 37
Ограничивающая сила, 520
Ориентированные ребра, 496
Оси
добавление в диаграмму, 159
соглашение о границах и отступах, 155
создание, 153
генерирование осей, 158
добавление меток осей, 165
объявление шкал, 158
Отображение
визуализаций с помощью Canvas, 628
кривых, 639
кругов, 637
линий, 634
простых фигур с помощью Canvas, 630
прямоугольников, 635
текста, 639

П

Передача части DOM под управление D3, 342
Передовые практики визуализации данных, 65
Пирамидальные диаграммы, 250
Площадь, 584
Подход с ориентацией
преимущественно на мобильные устройства, 377
преимущественно на настольные компьютеры, 377
Полярная система координат, 180
Порогов шкала, 677
Порядковая шкала, 680
Порядковые данные, 583
Последовательная квадратного корня шкала, 673
Последовательная логарифмическая шкала, 673
Последовательная симметричного логарифмирования шкала, 674
Последовательная степенная шкала, 673
Последовательная шкала, 672
квантилей, 678
Потоковый график, 65, 218
рисование, 218
Предметная область, 127, 665
Применение силовой компоновки к элементам на окружности, 605
Проблема площадных единиц, 545
Программы чтения с экрана, 405
Проекции, 535
Прямые, 151
линейные диаграммы, 166
генераторы линий, 168
интерполяция точек данных кривой, 170
области размаха, 173
создание осей, 153
генерирование осей, 158
соглашение о границах и отступах, 155
Пустой выбор, 118

Р

Радиальная шкала, 311, 670
Радиальные диаграммы с областями, 612
Радиальные столбчатые диаграммы, 616
Размещение городов на карте, 547
Распределения, 239
  гистограммы, 245
  пирамидальные диаграммы, 250
  скрипичные диаграммы, 265
  точечная шкала, 257
  ящичные диаграммы, 253
Растровое проецирование, 569
Расходящаяся логарифмическая шкала, 676
Расходящаяся степенная шкала, 675
Расходящаяся шкала, 674, 675
  квадратного корня, 675
  симметричного логарифмирования, 676
Ребра, 493
Рисование на холсте, 569
Руководящие принципы доступности веб-контента (Web Content Accessibility Guidelines, WCAG), 407

С

Самозакрывающиеся теги, 47
Сбор данных, 579
Свойства, 37
Связи, 453, 493, 584
Семантическое масштабирование, 558
Сетевые визуализации, 493
  дуговые диаграммы, 503
  запуск силовой компоновки, 509
  матрица смежности, 497
  подготовка данных, 495
  силовая компоновка, 509
Сетевые данные, 107
Сетка с 12 колонками, 389
Сила
  связи, 518
  столкновения, 514
  центрирования, 515
Силовая компоновка, 509
  оптимизация, 523
  остановка и перезапуск симуляции, 523
Силы позиционирования, 511
Символы, 584
Симметрического логарифмирования шкала, 671
Симуляция силовой компоновки, запуск, 509
Скобочная нотация, 61
Скрипичные диаграммы, 265
Сложные текстовые макеты SVG, 222
Смежность, 453
Смешанный режим отображения, Canvas и SVG, 641
Соглашение
  об отступах, 89
  о границах и отступах, 155
Создание
  адаптивных контейнеров SVG, 592
  адаптивных сеток SVG, 596
  основы проекта, 591
  переходов для повторного использования, 319
  плавных переходов, 289
  радиальных визуализаций, 599
Составные подсказки, 300
Составные проекции, 535
Составные столбчатые диаграммы, 215
Стековые диаграммы, 210
Стековые макеты, 194
Степенная шкала, 670
Степенные преобразования, 666
Стефанер, Мориц (Stefaner, Moritz), 274
Стили, 37, 91
Столбчатая диаграмма, 358
Стратегии взаимодействия с Canvas, 649

Т

Табличные данные, 104
Тамара Манцнер, 66
Текстовая информация, 409
  размер текста, 412
Точечная нотация, 61
Точечная шкала, 257, 679

У

Углы, 584

Узлы, 493
Узоры, 584
Упакованная пузырьковая диаграмма
  добавление всплывающих подсказок, 476
  добавление меток, 474
  макет, 465
  построение, 465
  рисование, 468
Упорядоченные списки, 428

Ф

Физические пиксели, 633
Фильтрация
  визуализаций, 282
    обработка событий, 284
    создание плавных переходов, 289
  диаграммы рассеяния, 313
  обработка событий, 284
Форматирование иерархических данных, 457
  файлы CSV, 458
  файлы JSON, 461
Фреймворки, 331
Фреймворки пользовательского интерфейса, 329
  вспомогательные библиотеки, 351
  гибридный подход, 366
  загрузка данных в проект React, 335
  контейнеры SVG, 338
  создание кривых, 360
  установка библиотеки D3 в проект React, 333
  Angular, 348
  React, 351
  Svelte, 349
Функции динамики, 289

Х

Хешам Эйсса, 66

Ц

Цвета
  информация должна передаваться не только с помощью цвета, 418
  контрастность, 416
Цилиндрические проекции, 535

Ш

Шаблонные литералы, 121
Шкалы, 127, 665
  временная шкала, 668
  выбор, 665
  идентичности шкала, 669
  интервальная шкала, 679
  квадратного корня шкала, 670
  квантилей шкала, 678
  квантования шкала, 677
  линейная шкала, 668
  логарифмическая шкала, 671
  непрерывные шкалы, 667
  порогов шкала, 677
  порядковая шкала, 680
  последовательная квадратного корня шкала, 673
  последовательная логарифмическая шкала, 673
  последовательная симметричного логарифмирования шкала, 674
  последовательная степенная шкала, 673
  последовательная шкала, 672
  последовательная шкала квантилей, 678
  последовательные, 672
  радиальная шкала, 670
  расходящаяся логарифмическая шкала, 676
  расходящаяся степенная шкала, 675
  расходящаяся шкала, 675
  расходящаяся шкала квадратного корня, 675
  расходящаяся шкала симметричного логарифмирования, 676
  расходящиеся, 674
  симметрического логарифмирования шкала, 671
  степенная шкала, 670
  точечная шкала, 679
  UTC, 669

Э

Эдвард Тафти, 66

Экосистема, 36
Экранные дикторы, 405
  доступность визуализированных данных для чтения, 433
  контейнеры SVG, 431
  пользовательские настройки клавиатуры не должны совпадать с комбинациями управления экранным диктором, 439
  предоставление таблиц данных, 435
  семантическая разметка, 427
Элайджи Микс, 35
Элементы
  добавление в выборку, 84
  заголовков, 427
Энди Кирк, 66
Эскиз макета, 589
Эхмель, Фабиан (Ehmel, Fabian), 274

Я

Ящичные диаграммы, 253
  позиционирование на графике, 256
  расчет квартилей с помощью квантильной шкалы, 254
  рисование, 260

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7, тел. **+7 (499) 322-19-38**,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Элайджи Микс, Энн-Мари Дюфур

D3.js в действии

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 58,99. Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**