Learn by doing: less theory, more results

Ogre 3D 1.7

Create real-time 3D applications using Ogre 3D from scratch

Beginner's Guide

Felix Kerger

[PACKT] PUBLISHING open source*
community experience distilled

Ogre 3D 1.7
Руководство новичка

Создание 3D-приложений реального времени с
использованием Ogre 3D с нуля

Автор: **Felix Kerger**

Перевод: **Striver**

Оглавление

1 Инсталляция Ogre3D....5
Загрузка и инсталляция Ogre 3D....5
Различные версии Ogre 3D SDK....5
Изучение SDK....6
Примеры Ogre 3D....7
Первое приложение с Ogre 3D....7
ExampleApplication....9
Загрузка первой модели....10
Итог....11
Дополнение переводчика: компиляция Ogre SDK в Linux....11
2 Граф сцены Ogre 3D....13
Создание узла сцены....13
Установка позиции узла сцены....17
Вращение узла сцены....18
Масштабирование узла сцены....20
Умный способ использования графа сцены 22
Различные пространства на сцене....24
Перемещение в локальном пространстве....27
Вращение в различных пространствах....29
Масштабирование в различных пространствах....30
Итог....31
3 Камера, свет и тень....32
Создание плоскости....32
Добавление точечного источника света....35
Добавление прожектора....36
Направленное освещение....39
Отсутствующая вещь....40
Добавление теней....41
Создание камеры....42
Создание порта просмотра....44
Итог....45
4 Получение ввода пользователя и использование Frame Listener....46
Подготовка сцены....46
Добавление движения на сцену....47
Модификация кода, чтобы движение зависело от времени, а не от кадра....50
Добавление поддержки ввода....51
Добавление движения модели....53
Добавление камеры....54
Добавление каркасного и точечного режимов рендера....56
Добавление таймера....57
Итог....58
5 Анимирование моделей в Ogre 3D....59
Добавление анимаций....59
Проигрывание двух анимаций в одно и то же время....61
Давайте немного пройдемся 63
Добавление сабель....65
Печать всех анимаций, имеющихся у модели....67
Итог....68
6 Менеджеры сцены....69
7 Материалы....69

8 Композиторы пост-обработки....69
9 Последовательность запуска Ogre3D....70
Запуск Ogre 3D....70
Добавление ресурсов....72
Использование resources.cfg....72
Создание класса приложения....74
Добавление FrameListener....76
Исследование функциональности FrameListener 77
Добавление ввода....81
Наш собственный главный цикл....82
Добавление камеры (снова)....84
Добавление композиторов....85
Добавление плоскости....87
Добавление управления пользователем....88
Добавление анимации....90
Итог....92
10 Системы частиц и расширение Ogre3D....93
Добавление системы частиц....93
Создание простой системы частиц....94
Немного больше параметров....96
Другие параметры....97
Включение её и выключение обратно....98
Добавление affector'ов....99
Изменение цветов....100
Двойное изменение....102
Ещё более сложные манипуляции с цветом....103
Добавление случайности....105
Deflector (Отклонятель)....106
Другие типы эмиттеров....107
Испускание из кольца....108
Наконец, мы хотели бы получить фейерверк....109
Расширение Ogre 3D....111
Итог....113
Конец....113

1 Инсталляция Ogre3D

Загрузка и установка новой библиотеки является первым шагом в её изучении и использовании.

В этой главе, мы должны сделать следующее:

- Загрузить и установить Ogre 3D
- Дать нашей среде разработки возможность работать с Ogre 3D
- Создать нашу первую сцену, отрендеренную Ogre 3D

Так что давайте займёмся этим.

Загрузка и инсталляция Ogre 3D

Первым шагом нам нужно установить и сконфигурировать Ogre 3D.

Время для действия — загрузка и установка Ogre 3D

Мы собираемся загрузить Ogre 3D SDK и установить его, чтобы мы могли работать с ним позже.

1. Перейдите на http://www.ogre3d.org/download/sdk
2. Загрузите подходящий пакет. Если Вам нужна помощь при выборе правильного пакета, взгляните на следующую секцию *Что произошло*.
3. Скопируйте инсталлятор в каталог, в котором бы Вы хотели видеть установленным ваш OgreSDK.
4. Дважды кликните по инсталлятору, он начнет с самораспаковки.
5. У вас должна появиться новая папка в вашем каталоге с именем, подобным OgreSDK_vc9_v1-7-1
6. Откройте эту папку. Она должна выглядеть подобно следующему снимку экрана:

Что же произошло?

Мы только что загрузили подходящий Ogre 3D SDK для нашей системы. Ogre 3D является кроссплатформенным движком рендера, так что есть много разных пакетов для этих различных платформ. После загрузки, мы распаковали Ogre 3D SDK.

Различные версии Ogre 3D SDK

Ogre поддерживает много различных платформ, и поэтому, существует множество различных пакетов, котрые мы можем загрузить. Ogre 3D имеет несколько сборок для Windows, одну для MacOSX, и один пакет Ubuntu. Есть также пакет для MinGW и для iPhone. Если Вы хотите, Вы можете загрузить исходный код и построить Ogre 3D самостоятельно. Эта глава сфокусирована на прекомпилированнном SDK для Windows, и на том, как сконфигурировать вашу среду разработки. Если Вы хотите использовать другую операционную систему, Вы можете посмотреть на Ogre 3D Wiki, которую можно найти по адресу http://www.ogre3d.org/wiki. Wiki содержит подробные уроки о том, как настроить среду разработки для множества различных платформ. Остальная часть книги является

полностью платформо-независимой, так что если Вы хотите использовать другую систему разработки, чувствуйте себя свободно, делая это. Это не повлияет на содержимое этой книги, только на конфигурацию и соглашения об окружении.

Изучение SDK

Прежде, чем мы начнем строить примеры, которые поставляются с SDK, давайте взглянем на SDK. Мы посмотрим на структуру SDK, имеющуюся на платформе Windows. На Linux или MacOS, структура может выглядеть по-другому. Сначала, мы открываем папку *bin*. Там мы увидим две папки, а именно: *debug* (отладка) и *release* (выпуск). Тот же истинно для каталога *lib*. Причина в том, что Ogre 3D SDK поставляется в отладочной и релизной сборках своих библиотек и динамически-связываемых/разделяемых библиотек. Это даёт возможность использовать отладочную сборку во время разработки, чтобы мы могли отлаживать наш проект. Когда мы выполним проект, мы слинкуем его с релизной сборкой, чтобы получить полную производительность Ogre 3D.

Когда мы откроем каталог *debug* или *release*, мы увидим много *dll*-файлов, несколько *cfg*-файлов, и два запускаемых файла (*exe*). Запускаемые файлы нужны для создателей контента, чтобы обновлять их файлы под новую версию Ogre 3D, и, следовательно, не важны для нас.

OgreMain.dll — наиболее важная DLL. Это — скомпилированный исходный код Ogre 3D, который мы загрузим позже. Все DLLки с именем, начинающимся с *Plugin_* — это плагины Ogre 3D, которые мы можем использовать с Ogre 3D. Плагины Ogre 3D — динамические библиотеки, которые добавляют новую функциональность к Ogre 3D, используя предлагаемые Ogre 3D интерфейсы. Это может быть практически всё, что угодно, но часто это используется для добавления функций, таких, как улучшенные системы частиц или новых менеджеров сцены. Что это такое, мы обсудим позже. Сообщество Ogre 3D создало множество плагинов, большинство из которых может быть обнаружено в wiki. SDK просто включает наиболее употребительные плагины. Позже в этой книге мы узнаем, как использовать некоторые из них. DLLки, имя которых начинается с *RenderSystem_*, что неудивительно, являются обёртками для различных систем визуализации (рендера), поддерживаемых Ogre 3D. В нашем случае, это — Direct3D9 и OpenGL. Дополнительно к этим двум системам, Ogre 3D также имеет Direct3D10, Direct3D11, и OpenGL ES (OpenGL для встроенных систем) системы визуализации.

Кроме исполняемых файлов и DLL-библиотек, у нас есть *cfg* файлы. Они являются файлами конфигурации, которые Ogre 3D может загружать при запуске. *Plugins.cfg* просто включает все плагины Ogre 3D, которые должны загружаться на запуске. Это обычно системы рендера Direct3D и OpenGL и несколько дополнительных менеджеров сцены. *quakemap.cfg* — это файл конфигурации, необходимый при загрузке уровня в формате карты *Quake3*. Нам этот файл не нужен, но примеры его используют.

resources.cfg содержит список всех ресурсов, таких как меши, текстуры или анимации, которые Ogre 3D должен загружать во время запуска. Ogre 3D может загружать ресурсы из файловой системы или из ZIP-архива. Если мы посмотрим на *resources.cfg*, мы увидим следующие строки:

```
Zip=../../media/packs/SdkTrays.zip
FileSystem=../../media/thumbnails
```

Zip= означает, что ресурс — это ZIP-архив, а *FileSystem=* означает, что мы хотим загрузить содержимое каталога. *resources.cfg* облегчает загрузку новых ресурсов, или изменение путей к ресурсам, так что он часто используется для загрузки ресурсов, особенно примерами Ogre 3D. Говоря о примерах, последний cfg-файл в каталоге — *samples.cfg*. Нам не нужно использовать этот *cfg*-файл. Кроме того, это — простой список со всеми примерами

Ogre 3D для загрузки в *SampleBrowser* (просмотрщик примеров). Но у нас пока нет *SampleBrowser*, так что давайте его построим.

Примеры Ogre 3D

Ogre 3D поставляется со множеством примеров, которые показывают все типы различных визуализируемых эффектов и техник, которые Ogre 3D может осуществить. Прежде, чем мы начнём работать с нашим приложением, мы взглянем на примеры, чтобы получить первое впечатление от возможностей Ogre 3D.

Время для действия — сборка примеров Ogre 3D

Чтобы получить первое впечатление от того, что Ogre 3D может делать, мы построим примеры и взглянем на них.

1. Перейдите в каталог *Ogre3D*.
2. Откройте solution-файл *Ogre3d.sln*.
3. Щелчок правой кнопкой по solution и выбор Build Solution.
4. Visual Studio должен теперь начать сборку примеров. Это может занять некоторое время, так что выпейте чашку чая, пока процесс компиляции не завершится.
5. Если всё прошло хорошо, войдите в папку *Ogre3D/bin*.
6. Выполните *SampleBrowser.exe*.
7. Вы должны увидеть на вашем экране следующее:
8. Попробуйте другие примеры, чтобы увидеть все отличные возможности, предлагаемые Ogre 3D.

Что произошло?

Мы построили примеры Ogre 3D, используя наш собственный Ogre 3D SDK. После этого, мы уверены в том, что имем рабочую копию Ogre 3D.

Популярная викторина — какие пост-эффекты показаны в примерах

Назовите по крайней мере пять различных пост-эффектов, которые показаны в примерах.

a) Bloom, Glass, Old TV, Black and White, and Invert
b) Bloom, Glass, Old Monitor, Black and White, and Invert
c) Boom, Glass, Old TV, Color, and Invert

Первое приложение с Ogre 3D.

В этой части мы создадим наше первое Ogre 3D - приложение , которое просто отрендерит одну 3D-модель.

Время для действия — начало проекта и конфигурация IDE

Как и для любой другой библиотеки, нам нужно сконфигурировать наше IDE прежде, чем мы можем использовать его с Ogre 3D.

1. Создайте новый пустой проект.
2. Создайте новый файл для кода и назовите его *main.cpp*.
3. Добавьте основную функцию *main*:

```
int main (void)
{
```

```
return 0;
}
```

4. Включите *ExampleApplication.h* в верхней части ниже исходного файла:
```
#include "Ogre\ExampleApplication.h":
```
5. Добавьте *PathToYourOgreSDK\include* в путь include (заголовочных файлов) вашего проекта.
6. Добавьте *PathToYourOgreSDK\boost_1_42* в путь include вашего проекта.
7. Добавьте *PathToYourOgreSDK\boost_1_42\lib* к вашему пути library (библиотек).
8. Добавьте новый класс к *main.cpp*.
```
class Example1 : public ExampleApplication
{
public:
void createScene()
{
}
};
```
9. Добавьте следующий код в начале вашей функции *main*:
```
Example1 app;
app.go();
```
10. Добавьте *PathToYourOgreSDK\lib\debug* к вашему пути library.
11. Добавьте *OgreMain_d.lib* к вашим линкуемым библиотекам.
12. Добавьте *OIS_d.lib* к вашим линкуемым библиотекам.
13. Скомпилируйте проект.
14. Установите вашему приложению рабочий каталог на *PathToYourOgreSDK\bin\debug*.
15. Запустите приложение. Вы должны увидеть диалог настройки Ogre 3D.
16. Нажмите OK и запустите приложение. Вы увидите черное окно. Нажмите *Escape*, чтобы выйти из приложения.

Что произошло?

Мы создали наше первое Ogre 3D - приложение. Чтобы скомпилировать, нам нужно настроить различные include- и library-пути, чтобы компилятор смог найти Ogre 3D.

В шагах 5 и 6, мы добавили два пути include к нашей среде сборки. Первый путь был к каталогу include Ogre 3D SDK, который содержит все файлы заголовков Ogre 3D и OIS. **OIS** представляет собой **Объектно-Ориентированную Систему Ввода (Object Oriented Input System)**, и это — библиотека ввода, которая используется классом *ExampleApplication* для обработки ввода пользователя. OIS не является частью Ogre 3D; это — автономный проект, и у него своя группа разработчиков. Он просто поставляется с Ogre 3D, поскольку *ExampleApplication* использует его, и, таким образом, пользователю не нужно дополнительно скачивать её для разрешения зависимости. *ExampleApplication.h* также находится в этом каталоге include. Поскольку Ogre 3D предлагает поддержку потоков (threading), нам нужно добавить каталог *boost* к нашим путям заголовочных файлов. В противном случае, мы не сможем собрать никакого приложения, использующего Ogre 3D. Если нужно, Ogre 3D можно собрать из исходных текстов с отключенной поддержкой потоков, и, таким образом, снять потребность в *boost*. И, при использовании *boost*, компилятор также должен быть способен слинковать библиотеки *boost*. Так что мы добавили каталог с библиотеками *boost* в наши

пути library (см. шаг 7).

На этапе 10 мы добавили *PathToYourOgreSDK\lib\debug* к нашему библиотечному пути. Как было сказано ранее, Ogre 3D поставляется с отладочными и релизными библиотеками. С этой строкой мы определяем использование отладочных библиотек, поскольку они предлагают лучшую поддержку отладки, если что-то случится в неправильном направлении. Когда мы захотим использовать релизную версию, мы должны изменить *lib\debug* на *\lib\release*. То же самое верно и для шагов 11 и 12. Там мы добавили *OgreMain_d.lib* и *OIS_d.lib* к нашим линкуемым библиотекам. Когда мы захотим использовать релизную версию, нам нужно будет добавить *OgreMain.lib* и *OIS.lib*. Оба файла, *OgreMain.lib* и *OgreMain_d.lib,* содержат информацию об интерфейсах Ogre 3D, и сообщают нашему приложению, что загрузить: *OgreMain.dll* или *OgreMain_d.dll*. Заметьте, что тоже самое верно и для системы ввода — *OIS.lib* или *OIS_d.lib*, они загружают *OIS_d.dll* или *OIS.dll*. Итак, мы линкуем (связываем) Ogre 3D и OIS динамически, что позволяет нам переключать DLL без повторной перекомпиляции нашего приложения, до тех пор, пока интерфейс библиотеки не изменится, и приложение и библиотеки DLL используют одни и те же версии библиотек времени выполнения. Это также подразумевает, что наше приложение всегда нуждается в загрузке DLL-файлов, так что мы должны убедиться, что оно сможет найти их. Это — одна из причин, почему мы установили рабочий каталог на этапе 14. Другая причина будет разъяснена в следующей секции.

ExampleApplication

Мы создали новый класс, *Example1*, который наследуется от *ExampleApplication*. *ExampleApplication* — класс, который поставляется с Ogre 3D SDK, и позволяет сделать изучение Ogre 3D легче, предлагая дополнительный слой абстракции поверх Ogre 3D. *ExampleApplication* стартует Ogre 3D для нас, загружает различные модели, которые мы можем использовать, и реализует простую камеру, чтобы мы могли перемещаться по нашей сцене. Чтобы использовать *ExampleApplication*, нам просто нужно наследоваться от него и переписать виртуальную функцию *createScene()*. Мы используем класс *ExampleApplication* сейчас, чтобы избавиться от большого количества работы, до того, пока мы не начнём хорошо понимать Ogre 3D. Позже, мы заменим *ExamplesApplication* по частям нашим собственным кодом.

В функции *main* мы создали новый экземпляр нашего прикладного класса и вызвали функциею *go()*, чтобы запустить приложение и загрузить Ogre 3D. При запуске, Ogre 3D загружает три конфигурационных файла — *Ogre.cfg*, *plugins.cfg*, и *resources.cfg*. Если мы используем отладочные (debug) версии, к имени каждого файла нужно добавить "_d". Это полезно, поскольку таким образом, мы можем иметь различные файлы конфигурации для отладки и релиза. *Ogre.cfg* содержит конфигурацию, которую мы выбрали в диалоге настройки, так что он может загрузить те же параметры настроек, чтобы избавить нас от ввода той же информации каждый раз, как мы запускаем наше приложение. *plugins.cfg* содержит список плугинов Ogre 3D, которые нужно загрузить. Наиболее важными являются плугины из группы *RenderSystem*. Они являются интерфейсом Ogre 3D для связи с OpenGL или DirectX, чтобы отрендерить свою сцену. *resources.cfg* содержит список ресурсов, которые Ogre 3D должен загрузить во время запуска. Ogre 3D SDK поставляется с большим количеством моделей и текстур, которые мы используем в этой книге, и *resources.cfg* указывает на их расположение. Если Вы посмотрите внутрь *resources.cfg*, Вы увидите, что пути в этом файле относительные. Это та причина, по которой нам нужно задать рабочий каталог.

Популярная викторина — какие библиотеки линкуются

1. Какие библиотеки Вам нужно прилинковать при использовании Ogre 3D в релизной конфигурации?
   1. OgreD3DRenderSystem.lib
   2. OgreMain.lib
   3. OIS.lib
2. Что мы должны будем изменить, когда мы захотим использовать отладочную версию сборки Ogre 3D?
   1. Добавить _debug после имени библиотеки
   2. Добавить _d к расширению файла
   3. Добавить _d после имени библиотеки

Загрузка первой модели

У нас есть базовое приложение с пустотой в нём, которое довольно скучно. Теперь мы загрузим модель, чтобы получить более интересную сцену.

Время для действия — загрузка модели

Загрузить модель легко. Нам просто нужно добавить две строки кода.

Добавьте следующие две строки в пустой метод *createScene()*:

1. Скомпилируйте снова ваше приложение.
```
Ogre::Entity* ent =
mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
mSceneMgr->getRootSceneNode()->attachObject(ent);
```
2. Запустите ваше приложение. Вы увидите небольшую зеленую фигуру после запуска приложения.
3. Управляйте камерой с помощью мыши и клавиш WASD, пока Вы не увидите зеленую фигуру получше.
4. Закройте приложение.

Что произошло?

Посредством *mSceneMgr->createEntity("MyEntity","Sinbad.mesh");*, мы сообщили Ogre 3D, что нам нужен новый экземпляр модели *Sinbad.mesh*. *mSceneMgr* — это указатель на *SceneManager* (Менеджер сцены) Ogre 3D, который создал для нас *ExampleApplication*. Для того, чтобы создать новую сущность (entity), Ogre нужно знать, какой файл модели использовать, и мы даём имя новому экземпляру. Важно, чтобы имя было уникальным; оно не может использоваться дважды. Если это случится, Ogre 3D возбудит исключение. Если мы не определим имя, Ogre 3D автоматически сгенерирует его за нас. Позже мы изучим это поведение более подробно.

Мы теперь имеем экземпляр модели, и, чтобы сделать его видимым, нам нужно подключить его к нашей сцене. Присоединить сущность довольно легко — просто записать следующую строку: *mSceneMgr->getRootSceneNode()->attachObject(ent);*

Это подключит сущность к нашей сцене, так что мы сможем увидеть её. И что мы видим? — Синбад, модель-талисман Ogre 3D. Мы увидим эту модель ещё много раз в этой книге.

Популярная викторина — ExampleApplication и как отображать модель

Опишите своими словами, как загрузить модель, и как сделать её видимой.

Итог

Мы узнали как Ogre 3D SDK организовано, какие библиотеки нам нужно прилинковать, и какой каталог нам нужен в путях наших заголовочных файлов (include). Также, мы бросили первый взгляд на класс *ExampleApplication*, и на то, как его использовать. Мы загрузили модель и отобразили её.

В подробностях, мы увидели:

- Какие файлы важны для разработки с Ogre 3D, как они взаимодействуют друг с другом, и какая у каждого из них цель
- Что *ExampleApplication* используется как эта вспомогательный класс, чтобы избавить нас от работы, и что случается во время запуска Ogre 3D
- Загрузка модели: Мы узнали как мы можем создать новый экземпряр модели с помощью *createEntity*, и один способ подключения нового экземпляра к нашей сцене

После этого введения в Ogre 3D, мы узнаем больше о том, как Ogre 3D организует сцены, и то, как мы можем манипулировать сценой, в следующей главе.

Дополнение переводчика: компиляция Ogre SDK в Linux

На моей домашней машине с Windows нет MS VisualC++, и, начиная перевод этой книги, я не планировал его ставить (для большинства собственных применений, в том числе и для Ogre, я предпочитаю использовать язык Python). Тем не менее, проверять работоспособность программ-примеров как-то надо. Я решил поставить Ogre SDK на свой маленький нетбук, на котором у меня стоит Meego Linux 1.0. Как уже написал автор книги, в прекомпилированном виде Ogre SDK поставляется только для Windows, MacOSX и Ubuntu. Так что мне пришлось компилировать SDK из исходных текстов. Это было не слишком сложно, но мне показалось, что это процесс заслуживает отдельного описания на случай, если кто-нибудь захочет его повторить. С исходными текстами Ogre 3D поставляется файл BuildingOgre.txt, в котором кратко расписаны процессы работы с CMake , удовлетворения зависимостей, настройки окружения, компиляции и инсталляции. В какой-то мере я здесь повторяю этот файл, но ведь на то он и перевод.

Кратко о Meego: дистрибутив, созданный для нетбуков, планшетов и (пока существует только 1 пример — Nokia N9) телефонов. Внутри, а именно с точки зрения пакетной системы сильно напоминает Федору, практически содран с неё. Пакеты от 12-й и 13-й Федоры без вопросов инсталлируются в Meego 1.0. Думаю, компиляция Ogre 3D в Федоре мало будет отличаться от того, что делал я.

Все действия по установке пакетов требуют рутовых прав, поэтому либо их надо выполнять от рута, либо дописывать перед каждой командой sudo и (иногда) вводить пароль.

Установка CMake. В файле BuildingOgre.txt написано, что для Убунты нужно в консоли набрать: *sudo apt-get install cmake-gui*. Для Meego, видимо, аналогичного пакета не создали, поэтому я сначала поставил cmake:

yum install cmake

Установился пакет *cmake-2.8.0-2.26.i586.rpm*. Потом я через RPMfind нашел близкий по

версии RPM-пакет cmake-gui, cmake-gui-2.8.1-1.i686.rpm, и установил его командой:

rpm -ihv --nodeps cmake-gui-2.8.1-1.i686.rpm

Установка зависимостей. Я не буду повторять команду для Убунты, написанную в BuildingOgre.txt (её там легко найти), нужные пакеты для Федоры отличались в названии (хотя бы суффиксом, у Убунты это -dev, а у Федоры -devel, но разница этим не ограничивалась). Вот так у меня должна была выглядеть команда установки, если бы я ставил эти пакеты скопом, а не по одному, выясняя имя каждого:

yum install freetype-devel zlib-devel cppunit-devel libXt-devel libXaw-devel libXxf86vm-devel libXrandr-devel mesa-libGLU-devel

Ещё 3 (на самом деле 6) пакета в репозитории Meego отсутствовали, я отдельно скачал их, найдя в RPMfind и установил через rpm:

rpm -ihv freeimage-3.10.0-3.fc12.i686.rpm freeimage-devel-3.10.0-3.fc12.i686.rpm ois-1.2.0-2.fc13.i686.rpm ois-devel-1.2.0-2.fc13.i686.rpm zziplib-0.13.49-8.fc12.i686.rpm zziplib-devel-0.13.49-8.fc12.i686.rpm

Я не смог найти пакетов boost-thread-devel и boost-date-time-devel для своей системы, хотя пакеты boost-thread и boost-date-time у меня уже были установлены. Весь boost целиком ставить мне не хотелось, поэтому я отказался от поддержки потоков. На одноядерном нетбуке они не слишком-то нужны. Также ввиду отсутствия дискретной видеокарты и поддержки шейдеров в нетбуке, я не стал ставить пакет Cg от NVIDIA.

Подготовка окружения сборки. Создал рядом с каталогом *ogre_src_v1-7-4*, содержащим исходные тексты, пустой каталог с именем *ogre-build*.

Работа CMake. Запустил cmake-gui, в поле ”Where is source code” определил положение каталога *ogre_src_v1-7-4* с исходными текстами, в поле ”Where to build the binaries” определил положение каталога *ogre-build*, где будет происходить сборка. Далее нужно первый раз нажать *Configure*. Оставил выбранными по-умолчанию *”Unix Makefiles”* и *”Use default native compilers”,* нажал *Finish*. После первоначальной конфигурации можно с помощью галочек выбрать, какие конкретно части Ogre 3D компилировать, а также выбрать каталог инсталляции, по-умолчанию он расположен в */usr/local*. В моём случае я поменял каталог инсталляции, но Вам, скорее всего, это не понадобится. Когда Вы решите, что Вас всё устраивает, нажимаем *Configure* ещё раз, и если всё нормально (т. е. нет проблем с зависимостями), то красный цвет фона должен смениться на обычный. Теперь можно нажать *Generate*.

Компиляция. Тут уже нет ничего необычного по сравнению с большинством других программ для Linux. Перешел в каталог *ogre-build* и выполнил там команду *make*. На моём нетбуке компиляция выполнялась полночи, надеюсь, ваш компьютер будет попроворнее.

Инсталляция. Тоже выполняется обычным *make install*. Так же в большинстве случаев требует рутовых прав, так что выполнять надо через *su* или *sudo*.

Использование Ogre 3D. Ogre 3D не является самостоятельным приложением, Вы будете использовать эту библиотеку при компиляции и запуске своих программ (или хотя бы примеров, поставляемых с Ogre 3D). Чтобы компиляция Вашей программы прошла успешно, нужно указать компилятору использовать библиотеки OGRE и OIS. Для себя я сделал скрипт с именем *complie.sh* такого вида:

g++ -B /путь-к-библиотеке-Ogre3D/ MyPrg.cpp 'pkg-config --cflags --libs OGRE OIS' -o MyPrg

Здесь название файла с текстом программы *MyPrg.cpp,* получаемый бинарный запускаемый файл *MyPrg.* Если Вы оставили каталог инсталляции по-умолчанию (т. е. */usr/local),* то, возможно, Вам часть *-B /путь-к-библиотеке-Ogre3D/* не понадобится.

2 Граф сцены Ogre 3D

Эта глава познакомит нас с понятием графа сцены, и с тем, как мы можем использовать его функции для создания сложных сцен.

В этой главе, мы изучим:

- Три основных операции в 3D-пространстве
- То, как организован граф сцены
- Различные 3D-пространства, в которых мы можем действовать

Так давайте начнём.

Создание узла сцены

В прошлой главе, *Глава 1, инсталляция Ogre 3D*, мы загрузили 3D-модель и добавили её к нашей сцене. Теперь мы узнаем как создать новый узел сцены, и подключить нашу 3D-модель к нему.

Время для действия — создание узла сцены с Ogre 3D

Мы собираемся использовать код из *Главы 1, Инсталляция Ogre 3D*, модифицировав его, чтобы создать новый узел сцены и подключить его к сцене. Мы будем следовать за этими шагами:

1. В старой версии нашего кода, у нас были следующие две строки в функции *createScene()*:
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
mSceneMgr->getRootSceneNode()->attachObject(ent);
```
2. Замените последнюю строку следующей:
```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
```
3. Затем добавьте следующие две строки; порядок этих двух строк не имеет отношения к получению сцены:
```
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```
4. Откомпилируйте и запустите приложение.
5. Вы должны увидеть тот же экран, который Вы получали, когда запускали приложение из *Главы 1*.

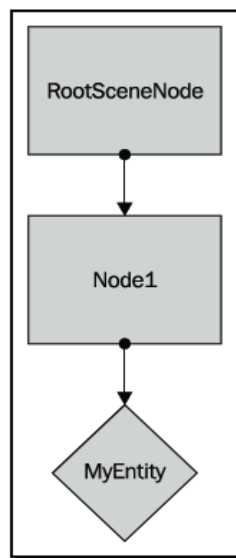
Что произошло?

Мы создали новый узел сцены с именем *Node1*. Затем мы добавили узел сцены к корневому узлу сцены. После этого, мы прикрепили нашу прежде созданную 3D-модель ко вновь созданному узлу сцены так, чтобы он стал видимым.

Как работать с RootSceneNode (корневым узлом сцены)

Вызов *mSceneMgr->getRootSceneNode()*, возвращает корневой узел сцены. Этот узел сцены является переменной-членом менеджера сцены. Когда мы хотим нечто сделать

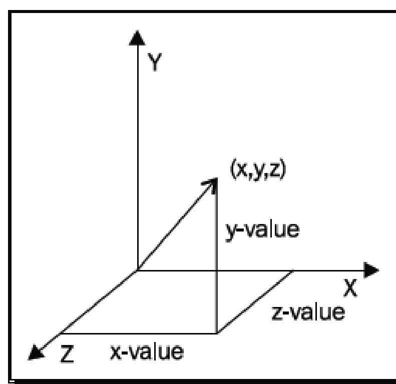
видимым, нам нужно подключить это к корневому узлу сцены или узлу, который является его ребенком или потомком. Короче говоря, должна быть цепь отношений родитель-ребенок от корневого узла до этого конкретного узла; в противном случае он не будет отрендерен. Как следует из названия, *RootSceneNode* (корневой узел сцены) является корнем сцены. Таким образом вся сцена будет, в каком то смысле, приложена к корневому узлу сцены. Ogre 3D использует так называемый граф сцены, чтобы её организовывать. Этот граф похож на дерево, он имеет один корень, корневой узел сцены, и каждый узел может иметь детей. Мы уже использовали эту характеристику когда мы вызывали *mSceneMgr->getRootSceneNode()->addChild(node);*. Там мы добавили созданный узел сцены в качестве ребенка корневого узла. Сразу вслед за этим, мы добавили другой тип ребенка к узлу сцены с помощью *node->attachObject(ent);*. Здесь мы добавили сущность (entity) к узлу сцены. У нас может быть несколько различных типов объектов, которые мы можем добавить к узлу сцены. Во-первых, у нас есть другие узлы сцены, которые можно добавить в качестве детей, и они могут сами иметь детей. Во-вторых, у нас есть сущности, которые мы хотим рендерить. Сущности не являются детьми и не могут сами иметь детей. Они — объекты данных, которые связаны с узлом, и о них можно думать, как о листах дерева. Существует много чего ещё, что мы можем добавить к изображению, например источники света, системы частиц, и так далее. Мы позже узнаем, что это такое, и как всё это использовать. Прямо сейчас, нам нужны только сущности. Наш текущий граф сцены выглядит похоже на следующее:



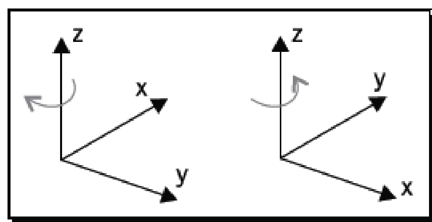
Первая вещь, которую мы должны понять — что такое граф сцены, и что он делает. Граф сцены используется, чтобы изобразить, как различные части сцены связаны друг с другом в 3D-пространстве.

3D-пространство

Ogre 3D — это графический 3D движок, так что нам нужно понимать некоторые основные концепции 3D. Основой конструирования в 3D является вектор, который представляют упорядоченной тройкой чисел (x,y,z).



Каждая позиция в 3D-пространстве может быть представлена такой тройкой, используя Евклидову систему координат для трех измерений. Важно знать, что есть различные типы систем координат в 3D-пространстве. Единственное различие между системами — ориентация оси и положительное направление вращения. Существуют две системы, которые широко используются, а именно: версия левой руки и правой руки. На следующем рисунке, мы видим обе системы — слева мы видим версию левой руки; с правой стороны мы видим версию правой руки.



Источник: http://en.wikipedia.org/wiki/File:Cartesian_coordinate_system_handedness.svg

Название левая и правая рука основываются на том, что ориентацию оси можно восстановить, используя левую и правую руки. Большой палец является осью x, указательный палец — это ось y, а средний палец — ось z. Нам нужно держать наши руки так, чтобы у нас был угол в 90 градусов между большим и указательным пальцами, а также между средним и указательным пальцем. При использовании правой руки, мы получаем систему координат правой руки. При использовании левой руки, мы получаем левостороннюю версию.

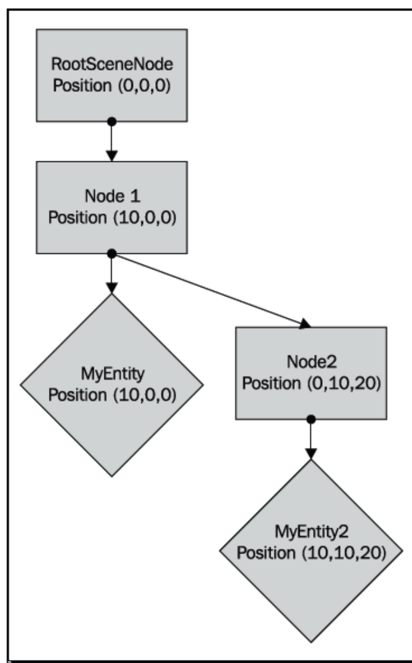
Ogre 3D использует систему правой руки, но поворачивает её так, чтобы положительная часть оси x была направлена вправо, а отрицательная часть налево. Ось y направлена вверх, а ось z указывает из экрана, и такая система известна как **y-вверх соглашение (y-up convention)**. Поначалу это звучит раздражающе, но мы скоро научимся думать в этой системе координат. Вебсайт http://viz.aset.psu.edu/gho/sem_notes/3d_fundamentals/html/3d_coordinates.html содержит довольно хорошее, основанное на иллюстрациях, объяснение различных координатных систем, и того, как они соотносятся друг с другом.

Граф сцены

Граф сцены — одно из наиболее используемых понятий в графическом программировании. Говоря по-простому, это способ хранить информацию о сцене. Мы уже обсудили, что граф сцены имеет корень и организован подобно дереву. Но мы не затрагивали наиболее важную функцию графа сцены. Каждый узел графа имеет список своих детей, а также преобразование (transformation) в пространстве 3D. Преобразование формируется из трех аспектов, а именно: позиции, вращения, и масштаба. Позиция — это тройка (x,y,z), которая, очевидно, описывает позицию узла на сцене. Вращение запоминается с использованием кватерниона (quaternion), математического понятия для хранения вращений в

3D-пространстве, но мы можем думать о вращении, как о единичной величине с плавающей точкой для каждой оси, описывающей то, как узел повёрнут, с использованием радианов в качестве единиц измерения. С масштабированием совсем просто; снова, оно использует тройку (x,y,z), и каждая её часть просто является величиной, на которую масштабируется соответствующая ось.

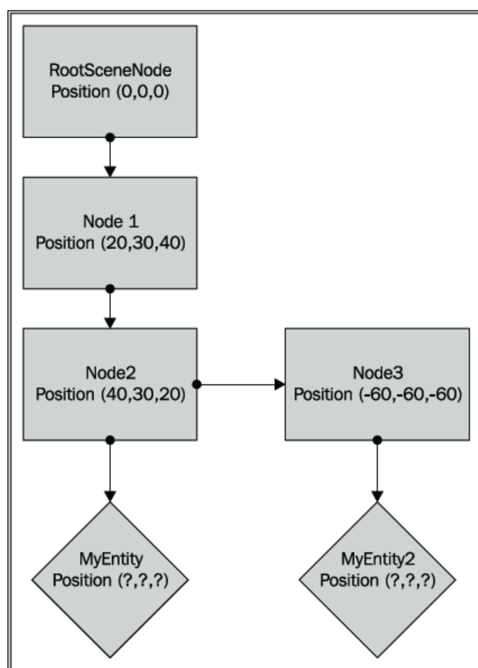
Важный момент по поводу графа сцены состоит в том, что преобразование производится относительно родительского узла. Если мы изменим ориентацию родителя, на детей также будут воздействовать эти изменения. Когда мы перемещаем родителя на 10 единиц вдоль оси x, все дети также будут перемещаться на 10 единиц вдоль оси x. Конечная ориентация каждого ребенка вычисляется с учетом ориентации всех родителей. Этот факт станет более ясным с помощью следующей диаграммы.



Позиция **MyEntity** на этой сцене будет **(10,0,0)**, а **MyEntity2** будет находится в **(10,10,20)**. Давайте попробуем это в Ogre 3D.

Популярная викторина — поиск позиции узлов сцены

1. Посмотрите на следующее дерево и определите конечные позиции **MyEntity** и **MyEntity2**:
    a. MyEntity(60,60,60) и MyEntity2(0,0,0)
    b. MyEntity(70,50,60) и MyEntity2(10,-10,0)
    c. MyEntity(60,60,60) и MyEntity2(10,10,10)



Установка позиции узла сцены

Теперь мы попытаемся настроить сцену в соответствии с диаграммой на предыдущем рисунке.

Время для действия — установка позиции узла сцены

1. Добавьте эту новую строку после создания узла сцены:
```
node->setPosition(10,0,0);
```
2. Для того, чтобы создать вторую сущность, добавьте эту строку в конце функции *createScene()*:
```
Ogre::Entity* ent2 = mSceneMgr->createEntity( "MyEntity2", "Sinbad.mesh");
```
3. Затем создайте второй узел сцены:
```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
```
4. Добавьте второй узел к первому:
```
node->addChild(node2);
```
5. Установите позицию второго узла:
```
node2->setPosition(0,10,20);
```
6. Подключите вторую сущность ко второму узлу:
```
node2->attachObject(ent2);
```
7. Скомпилируйте программу, и Вы должны увидеть два экземпляра Синбада:

Что произошло?

Мы создали сцену, которая соответствует предыдущей диаграмме. Первая новая функция, которую мы использовали, была в 1 шаге. Легко догадаться, что функция *setPosition(x,y,z)* задаёт позицию узла заданной тройкой. Имейте в виду, что эта позиция относительно родителя. Мы захотели, чтобы *MyEntity2* находился в *(10,10,20)*, поскольку мы добавили узел *node2*, который содержит *MyEntity2*, к узлу сцены, который уже был в позиции *(10,0,0)*. Нам только нужно задать позицию node2 в *(0,10,20)*. Когда обе позиции комбинируются, *MyEntity2* окажется в *(10,10,20)*.

Популярная викторина — играем с узлами сцены

1. У нас есть узел сцены *node1*, находящийся в *(0,20,0)*, и у нас есть узел сцены-потомок *node2*, к которому приложена сущность. Если мы хотим, чтобы сущность была изображена в *(10,10,10)*, в какой позиции нам нужно установить *node2*?
   a. (10,10,10)
   b. (10,-10,10)
   c. (-10,10,-10)

Have a go hero — Добавление Синбада

Добавьте третий экземпляр Синбада, и пусть он будет изображён в позиции (10,10,30).

Вращение узла сцены

Мы уже знаем, как установить позицию узла сцены. Теперь, мы узнаем, как вращать узел сцены, и другим способом модифицируем позицию узла сцены.

Время для действия — вращение узла сцены

Мы используем предыдущий код, но создадим полностью новый код для функции *createScene()*.

1. Удалите весь код из функции *createScene()*.
2. Сначала создайте экземпляр *Sinbad.mesh*, затем создайте новый узел сцены. Установите позицию узла сцены в (10,10,0), в конце подключите сущность к узлу, и добавьте узел к корневому узлу сцены в качестве ребенка:
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity", "Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10,10,0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```
3. Снова, создайте новый экземпляр модели, также новый узел сцены и установите его в (10,0,0):
```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
node2->setPosition(10,0,0);
```
4. Теперь добавьте следующие две строки, чтобы повернуть модель и подключить сущность к узлу сцены:
```
node2->pitch(Ogre::Radian(Ogre::Math::HALF_PI));
node2->attachObject(ent2);
```
5. Сделайте снова то же самое, но на этот раз используйте функцию *yaw* (отклонять) вместо функции *pitch* (наклонять) и функцию *translate* вместо функции *setPosition*:

```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = mSceneMgr->createSceneNode("Node3",);
node->addChild(node3);
node3->translate(20,0,0);
node3->yaw(Ogre::Degree(90.0f));
node3->attachObject(ent3);
```

6. И ещё раз с функцией *roll* (прокручивать) вместо *yaw* или *pitch*:

```
Ogre::Entity* ent4 = mSceneMgr->createEntity("MyEntity4","Sinbad.mesh");
Ogre::SceneNode* node4 = mSceneMgr->createSceneNode("Node4");
node->addChild(node4);
node4->setPosition(30,0,0);
node4->roll(Ogre::Radian(Ogre::Math::HALF_PI));
node4->attachObject(ent4);
```

7. Скомпилируйте и запустите программу, и Вы должны увидеть следующую картинку:

Что произошло?

Мы четыре раза повторили код, который у нас был до этого, и при этом изменяли некоторые небольшие детали. В первом повторении нет ничего специфического. Это тот же код, который мы использовали ранее, и на этот экземпляр модели будет референсным, чтобы видеть, что случилось с тремя другими экземплярами, которые мы делали далее.

На шаге 4 мы добавили одну следующую дополнительную строку:

```
node2->pitch(Ogre::Radian(Ogre::Math::HALF_PI));
```

Функция *pitch(Ogre::Radian(Ogre::Math::HALF_PI))* вращает узел сцены вокруг оси x. Как сказано ранее, эта функция принимает параметр в радианах, и мы использовали половину pi, что означает вращение на девяносто градусов.

На 5 этапе, мы заменили вызов функции *setPosition(x,y,z)* на *translate(x,y,z)*. Различие между *setPosition(x,y,z)* и *translate(x,y,z)*, в том, что *setPosition* устанавливает позицию — здесь нет сюрпризов. *translate* добавляет полученную величину к позиции узла сцены, так что она перемещает узел относительно его текущей позиции. Если узел сцены расположен в (10,20,30) и, мы вызываем *setPosition(30,20,10)*, тогда узел займёт позицию (30,20,10). С другой стороны, если мы вызываем *translate(30,20,10)*, узел встанет на позицию (40,40,40). Это — небольшое, но важное, отличие. Обе функции могут быть полезными, если их использовать в правильных обстоятельствах, например, когда мы хотим спозиционировать объект на сцене, мы должны использовать функцию *setPosition(x,y,z)*. В то же время, когда мы хотим переместить уже установленный на сцене узел, мы должны использовать *translate(x,y,z)*.

Также, мы заменили *pitch(Ogre::Radian(Ogre::Math::HALF_PI))* на *yaw(Ogre::Degree(90.0f))*. Функция *yaw()* вращает узел сцены вокруг оси y. Вместо *Ogre::Radian()* мы использовали *Ogre::Degree()*. Конечно, *pitch* и *yaw* все еще должны использовать углы в радианах. Тем не менее, Ogre 3D предлагает класс *Degree()*, который имеет оператор приведения, так что компилятор может автоматически переводить градусы в

радианы. Следовательно, программист свободен в использовании радианов или градусов при вращении узлов сцены. Обязательное использование классов даёт уверенность, что всегда известно, какие единицы использовались, чтобы предотвратить неразбериху и возможные источники ошибок.

Шаг 6 вводит последний из трех различных имеющихся функций поворота узла сцены, а именно — *roll()*. Эта функция вращает узел сцены вокруг оси z. Снова, мы могли бы использовать *roll(Ogre::Degree(90.0f))* вместо *roll(Ogre::Radian(Ogre::Math::HALF_PI))*.

Программа при запуске показывает модель в первоначальном положении и все три возможных вращения. Левая модель без вращения, модель правее от неё повёрнута вокруг оси x, 3-я модель повёрнута вокруг оси y, и модель справа повёрнута вокруг оси z. Каждый из этих примеров показывает эффект от различных функций вращения. Короче говоря, *pitch()* поворачивает вокруг оси x, *yaw()* вокруг y, и *roll()* вокруг z. Мы можем использовать или *Ogre::Degree(degree)* или *Ogre::Radian(radian)*, чтобы задать угол, на который мы хотим повернуть.

Популярная викторина — вращение узла сцены

Какие три функции вращают узлы сцены?

a. pitch, yawn, roll
b. pitch, yaw, roll
c. pitching, yaw, roll

Have a go hero — использование Ogre::Degree

Переделайте код, который мы писали в предыдущей секции таким образом, чтобы каждый случай использования *Ogre::Radian* был заменен *Ogre::Degree* и наоборот, а вращение оставалось всё тем же.

Масштабирование узла сцены

Мы уже разобрались с двумя из трех основных действий, которые мы можем использовать, манипулируя нашим графом сцены. Теперь подошло время для последнего, а именно, масштабирования.

Время для действия — масштабирование узла сцены

Снова, мы начинаем с того же блока кода, который мы использовали до этого.

1. Удалите весь код из функции *createScene()* и вставьте следующий код:
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10,10,0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```
2. Опять, создайте новую сущность:
```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");
```
3. Теперь мы используем функцию, которая создает узел сцены и добавляет его автоматически в качестве ребенка. Затем мы делаем то же, что мы делали прежде:
```
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10,0,0);
node2->attachObject(ent2);
```
4. Теперь, после функции *setPosition()*, вызовите следующую строку для

масштабирования модели:
```
node2->scale(2.0f,2.0f,2.0f);
```

5. Создайте новую сущность:
```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
```

6. Теперь мы вызываем ту же функцию, что и на этапе 3, но с дополнительным параметром:
```
Ogre::SceneNode* node3 = node->createChildSceneNode("node3",
Ogre::Vector3(20,0,0));
```

7. После вызова функции, вставьте эту строку, чтобы отмасштабировать модель:
```
node3->scale(0.2f,0.2f,0.2f);
```

8. Скомпилируйте программу и запустите её, и Вы должны увидеть следующее изображение:

Что произошло?

Мы создали сцену с масштабированными моделями. Ничего нового не было до 3 шага. Затем мы использовали новую функцию, а именно: *node->createChildSceneNode("node2")*. Эта функция является функцией-членом класса узла сцены и создает новый узел с полученным именем и добавляет его непосредственно в качестве ребенка к узлу, у которого была вызвана функция. В этом случае, node2 был добавлен, как ребенок узла node.

На этапе 4, мы использовали функцию узла сцены *scale()*. Эта функция принимает тройку чисел (x,y,z), которая указывает, как узел сцены должен быть масштабирован. x, y, и z — показатели. (0.5,1.0,2.0) означает, что узел сцены должен быть уменьшен вдвое по оси x, оставлен таким же по оси y, и удвоен по оси z. Конечно, если подойти к вопросу строго, узел сцены не может масштабирован; он только содержит метаданные, которое не будут рендериться. Более точным будет сказать, что каждый визуализируемый объект, приложенный к этому узлу, должен быть масштабирован вместо самого узла сцены. Узел — это только то, что содержит ссылки на все присоединённые потомки и визуализируемые объекты.

На шаге 6 мы снова использовали функцию *createChildSceneNode()*, но на этот раз с большим количеством параметров. Второй параметр, который принимает эта функция, тройка (x,y,z), которую мы так часто использовали. Ogre 3D имеет собственный класс для неё, называемый *Ogre::Vector3*. Кроме хранения тройки, этот класс предлагает функции, которые осуществляют основные действия с трехмерными векторами в линейной алгебре. Этот вектор описывает смещение, на которое должен быть сдвинут создаваемый узел сцены. *createChildSceneNode()* можно использовать, чтобы заменить следующие строки кода:

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
```

или даже

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node->addChild(node2);
node2->setPosition(20,0,0);
```

Последнюю часть кода можно заменить на

```
Ogre::SceneNode* node2 = node->createChildSceneNode("Node2",
Ogre::Vector3(20,0,0));
```

Если мы пропустим параметр *Vector3*, мы можем заменить первый кусок кода. Существует больше версий этой функции, которые мы используем позже. Если Вы не хотите ждать, взгляните на документацию Ogre 3D по адресу http://www.ogre3d.org/docs/api/html/index.html.

Кроме *scale()*, существует также функция *setScale()*. Различие между этими функциями такое же, как и между *setPosition()* и *translate()*.

Популярная викторина — создание узлов сцены - потомков

1. Назовите два различных способа вызова *createChildSceneNode()*.
2. Как следующая строка могла бы быть заменена без использования *createChildSceneNode()*?

```
Ogre::SceneNode* node2 = node->createChildSceneNode("node1",
Ogre::Vector3(10,20,30));
```

Эту строку можно заменить тремя строками. Первая создает узел сцены, вторая его перемещает, и третья подключает его к узлу.

```
Ogre::SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
node2->translate(Ogre::Vector3(10,20,30));
node->addChild(node2);
```

Have a go hero — использование createChildSceneNode()

Переработайте (refactor) весь код, который Вы написали в этой главе, чтобы использовать *createChildSceneNode()*.

Умный способ использования графа сцены

В этой секции мы узнаем, как мы можем использовать характеристики графа сцены, чтобы облегчить некоторые задачи. Это также расширит наши знания о графе сцены.

Время для действия — построение дерева с использованием узлов сцены

На этот раз мы собираемся использовать ещё одну модель помимо Синбада — ниндзя.

1. Удалите весь код из функции *createScene()*.
2. Создайте Синбада точно также, как мы всегда делали:

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(10,10,0);
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

3. Теперь создайте ниндзю, который повсюду будет следовать за Синбадом:

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntitysNinja",
"ninja.mesh");
```

```
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10,0,0);
node2->setScale(0.02f,0.02f,0.02f);
node2->attachObject(ent2);
```

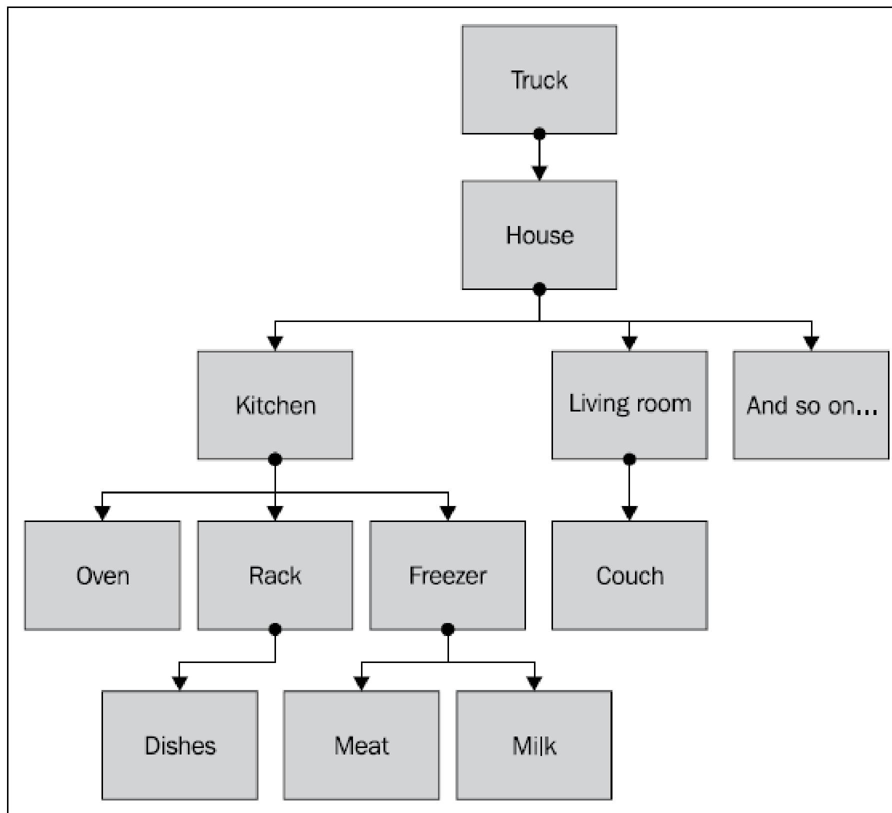
4. Скомпилируйте и запустите приложение. Когда Вы посмотрите поближе на Синбада, Вы увидите зеленого ниндзю со стороны его левой руки.

5. Теперь измените позицию в (40,10,0):
```
node->setPosition(40,10,0);
```
6. И поверните модель на 180 градусов вокруг оси x:
```
node->yaw(Ogre::Degree(180.0f));
```
7. Скомпилируйте программу и запустите её.
8. Вы должны увидеть, что ниндзя все ещё находится по левую руку от Синбада, а Синбад повёрнут.

Что произошло?

Мы создали пронырливого ниндзю, который следует за каждым шагом Синбада. Мы сделали это возможным, поскольку добавили узел, к которому приложена модель ниндзи так, чтобы он был ребенком узла сцены, к которому приложен Синбад. Когда мы перемещали Синбада, мы использовали его узел сцены, так что каждая трансформация, которую мы делали, также применялась к ниндзе, поскольку этот узел сцены является ребенком модифицируемого узла, и как сказано ранее, трансформация родителя передаётся всем детям. Этот факт об узлах сцены чрезвычайно полезен, чтобы создавать следующие друг за другом модели или сложные сцены. Скажем, если бы мы захотели создать грузовик, который несет на себе целый дом, мы могли бы создать дом, используя много различных моделей и узлов

сцены. В конечном счете, у нас был бы узел сцены с прикреплённым домом, а предметы его интерьера добавлены к нему в качестве детей. Теперь, когда мы хотим переместить дом, мы просто подключаем узел дома к узлу грузовика, и при перемещении грузовика весь дом перемещается вместе с ним.



Стрелки показывают направление, в котором преобразования распространяются вдоль графа сцены.

Популярная викторина — ещё больше о графе сцены

Как информация преобразований проходит в графе сцены?

a. От листов до корня

b. От корня до листов

Have a go hero — добавление следующего ниндзи

Добавьте второго ниндзю к сцене, который будет следовать за первым ниндзей.

Различные пространства на сцене

В этой части мы узнаем, что есть различные пространства на сцене, и как мы можем использовать эти пространства.

Время для действия — перемещения в пространстве Мира

Мы собираемся переместить объект способом, отличающимся от тех, что мы использовали.

1. Снова, начните с пустой функции *createScene()*; удалите весь код, который есть в этой функции.
2. Создайте референсный объект:

```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(0,0,400);
node->yaw(Ogre::Degree(180.0f));
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```

3. Создайте два новых экземпляра модели и переместите каждый на (0,0,10):

```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(10,0,0);
node2->translate(0,0,10);
node2->attachObject(ent2);

Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setPosition(20,0,0);
node3->translate(0,0,10);
node3->attachObject(ent3);
```

4. Скомпилируйте и запустите приложение. Управляйте камерой, пока Вы не увидите предыдущие модели примерно так:

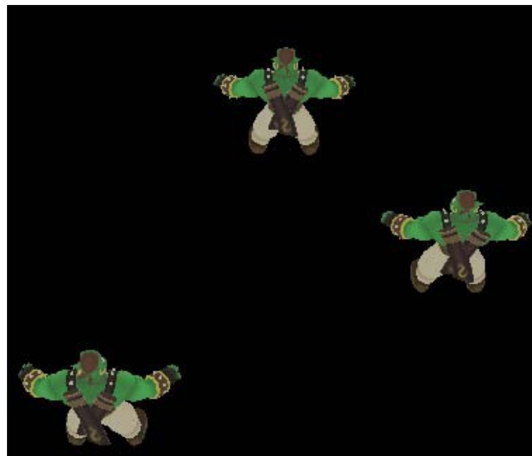
5. Замените строку:

```
node3->translate(0,0,10);
```

на

```
node3->translate(0,0,10,Ogre::Node::TS_WORLD);
```

6. Снова, скомпилируйте и запустите приложение и настройте камеру как предыдущий раз.



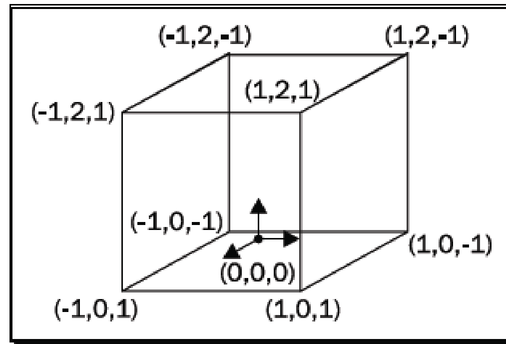
Что произошло?

Мы использовали новый параметр функции *translate()*. Результатом стало то, что левая

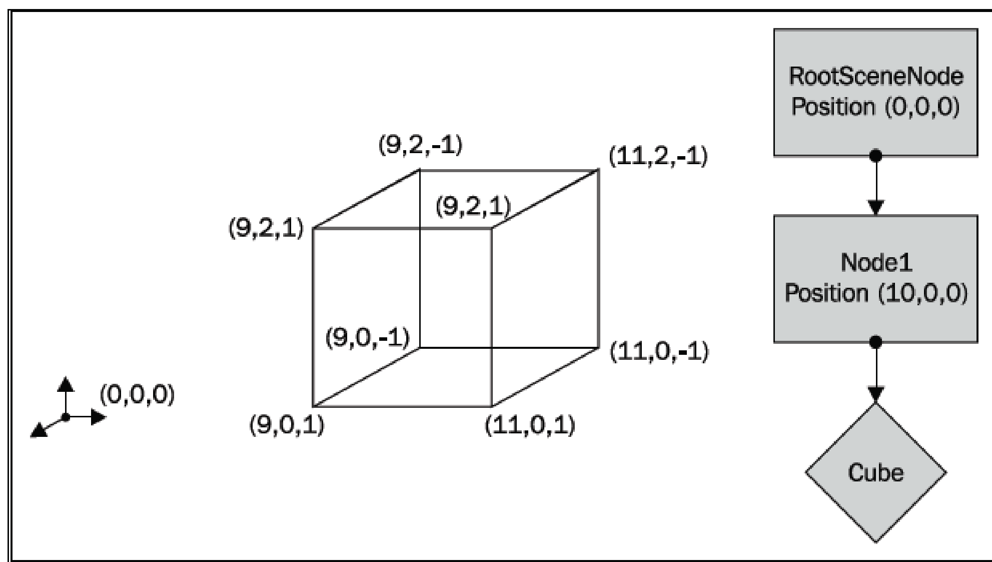
модель на сцене переместилась в другом направлении от средней модели.

Различные пространства в 3D-сцене

Причина, по которой модель переместилась иначе в том, что с помощью *Ogre::Node::TS_WORLD* мы сообщили функции *translate()*, что перемещение должно происходить в мировом пространстве, а не в пространстве родителя, что установлено по-умолчанию. Существует три типа пространства у 3D-сцены: мировое (world) пространство, пространство родителя (parent), и локальное (local) пространство. В локальном пространстве определена сама модель. Куб состоит из восьми точек и может быть описан как на следующей диаграмме:



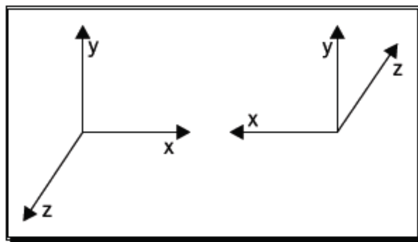
Черная точка является началом координат локального пространства. Каждая точка куба описана относительно начала координат. Когда сцена рендерится, куб нужен в мировом пространстве. Чтобы получить куб в мировом пространстве, все родительские преобразования графа сцены применяются к этим точкам. Скажем, куб приложен к узлу сцены, который добавлен к корневому узлу сцены с *translate(10,0,0)*. Тогда куб в мировом пространстве должен выглядеть так:



Различие между двумя кубами в том, что начало координат переместилось, или, чтобы быть более точным, куб переместился дальше от начала координат.

Когда мы вызываем функцию *translate()*, куб перемещается в родительском пространстве, если используемое пространство не определено, подобно тому, как мы делали на этапе 5. Пока ни один из предков куба не был повёрнут, *translate()*, ведет себя так же в мировом пространстве, как это было бы при использовании родительского или локального пространства. Это верно, поскольку меняется только позиция начала координат, а не ориентация осей. Когда мы, например, перемещаем куб на (0,0,10) единиц, не имеет

значения, где находится начало координат — пока оси координатных систем ориентированы одинаково, результат процесса перемещения не будет меняться. Тем не менее, если родителя повернуть, это перестанет быть истиной. При повороте родителя также повернутся оси начала координат, что изменит результат *translate(0,0,10)*.



Левая координатная система не повёрнута, и (0,0,10) означает перемещение куба на 10 единиц ближе к зрителю. Дело в том, что ось z направлена из экрана наружу. С поворотом на 180 градусов, (0,0,10) означает перемещение куба на 10 единиц вдаль от зрителя, поскольку теперь ось z указывает в экран.

Мы видим как важно, в каком пространстве мы описываем функцию *translate()*, чтобы получить желаемый эффект. Мировое пространство всегда имеет одну и ту же ориентацию осей. Чтобы быть точным, мировое пространство использует левостороннюю координатную систему. Родительское пространство использует координатную систему, где приложены вращения всех родителей вверх по цепочке. Локальное пространство включает все вращения, от своего собственного узла сцены и всех родительских. По-умолчанию *translate()* должно использовать родительское пространство. Это позволяет нам вращать сам узел, не изменяя направление перемещения узла, когда используется *translate()*. Но бывают случаи, когда нам нужно перемещать в пространстве, отличном от родительского. В таких случаях, мы можем использовать второй параметр функции *translate()*. Второй параметр определяет пространство, в котором мы хотим произвести перемещение. В нашем коде, мы использовали *Ogre::Node::TS_WORLD*, чтобы перемещать модель в мировом пространстве, которое инвертировало направление движения модели, поскольку мы повернули узел на 180 градусов, и таким образом, перевернули направление осей x и z. Кроме того, взгляните на изображение, чтобы увидеть эффект.

Перемещение в локальном пространстве

Мы уже видели эффект перемещения в родительском и мировом пространстве. Теперь мы будем перемещать в локальном и родительском пространстве, чтобы увидеть и глубже понять различия между пространствами.

Время для действия — перемещение в локальном и родительском пространствах

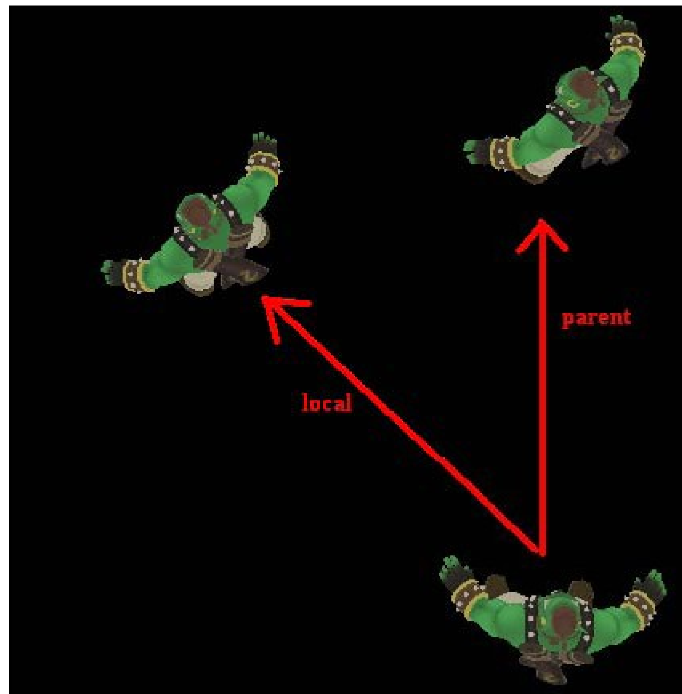
1. Снова очистите функцию *createScene()*.
2. Добавьте референсную модель; на этот раз мы переместим её ближе к нашей камере, так что нам не придётся перемещать камеру так сильно:
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","Sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
node->setPosition(0,0,400);
node->yaw(Ogre::Degree(180.0f));
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```
3. Добавьте вторую модель, поверните её на `45` градусов вокруг оси `y` и переместите на `(0,0,20)` единиц в родительском пространстве:
```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","Sinbad.mesh");
```

```
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->yaw(Ogre::Degree(45));
node2->translate(0,0,20);
node2->attachObject(ent2);
```

4. Добавьте третью модель, и также поверните её `45` градусов вокруг оси `y` и переместите на `(0,0,20)` единиц в локальном пространстве:

```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->yaw(Ogre::Degree(45));
node3->translate(0,0,20,Ogre::Node::TS_LOCAL);
node3->attachObject(ent3);
```

5. Скомпилируйте и запустите приложение. Затем подвигайте камерой снова, чтобы посмотреть на модели сверху.



Что произошло?

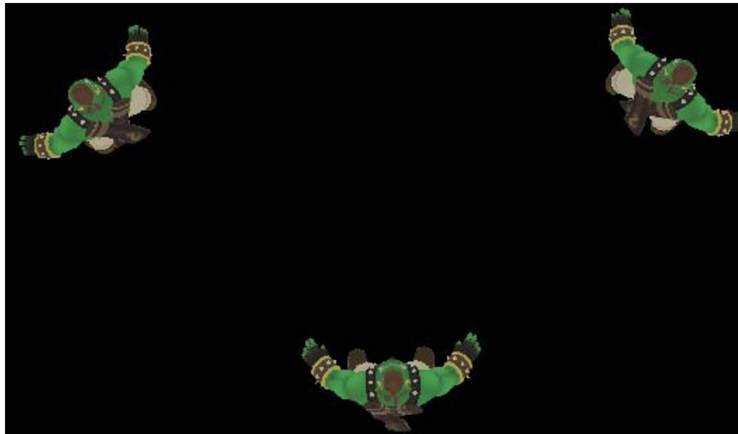
Мы создали нашу референсную модель и затем добавили две модели, которые повернули на 45 градусов вокруг оси y. Затем мы переместили обе модели на (0,0,20), одну модель в родительском пространстве, работающем по-умолчанию, и другую модель в локальном пространстве. Модель, которую мы перемещали в родительском пространстве сместилась прямо по оси z. Но поскольку мы повернули модели вокруг оси y, то модель, которую мы перемещали в локальном пространстве, сместилась из-за вращения и передвинулась вверх и влево на рисунке. Давайте повторим это. Когда мы вызываем *translate*, настройкой по-умолчанию является родительское пространство, это означает, что все вращения, за исключением вращения того узла сцены, который мы перемещаем, используются для расчетов. При использовании мирового пространства вообще никакое вращение не принимается во внимание. При перемещении используется мировая система координат. При перемещении в локальном пространстве, учитывается каждое вращение, даже вращение своего узла.

Популярная викторина — Ogre 3D и пространства

Назовите три различных пространства, которые знает Ogre 3D.

Have a go hero — добавление симметрии

Измените вращение и перемещение *MyEntity2*, чтобы сделать изображение симметричным. Убедитесь, что Вы используете правильное пространство; в противном случае, будет трудно создать симметрию. Вот как это должно выглядеть:



Вращение в различных пространствах

Мы уже видели, как различные пространства используются при перемещении; теперь мы сделаем тот же самое с вращением.

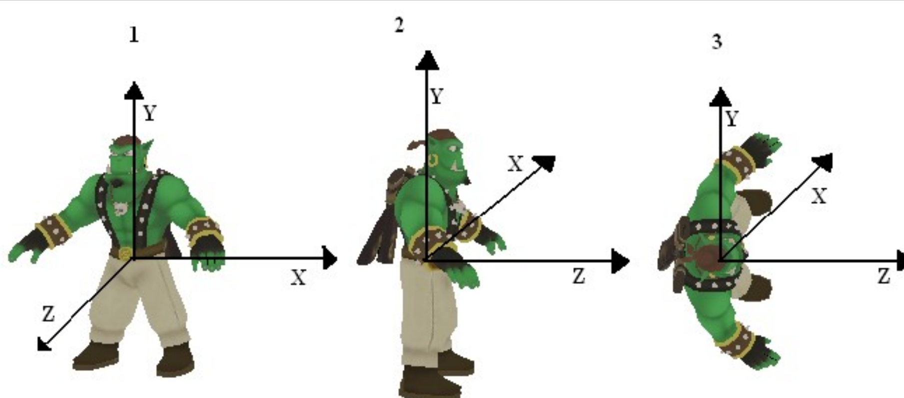
Время для действия — вращение в различных пространствах

На этот раз мы собираемся вращать, используя различные пространства, следующим образом:

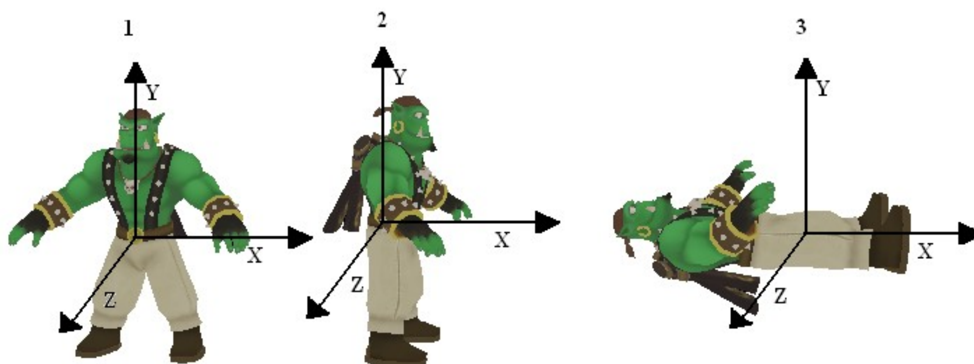
1. И снова, мы начнем с очистки функции *createScene()*, так что удалите весь код в этой функции.
2. Добавьте референсную модель:
```
Ogre::Entity* ent = mSceneMgr->createEntity("MyEntity","sinbad.mesh");
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
node->attachObject(ent);
```
3. Добавьте вторую модель и поверните её нормальным способом:
```
Ogre::Entity* ent2 = mSceneMgr->createEntity("MyEntity2","sinbad.mesh");
Ogre::SceneNode* node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("Node2");
node2->setPosition(10,0,0);
node2->yaw(Ogre::Degree(90));
node2->roll(Ogre::Degree(90));
node2->attachObject(ent2);
```
4. Добавьте третью модель и поверните её, используя мировое пространство:
```
Ogre::Entity* ent3 = mSceneMgr->createEntity("MyEntity3","Sinbad.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setPosition(20,0,0);
node3->yaw(Ogre::Degree(90),Ogre::Node::TS_WORLD);
node3->roll(Ogre::Degree(90),Ogre::Node::TS_WORLD);
node3->attachObject(ent3);
```
5. Скомпилируйте и запустите приложение.

Что произошло?

Как всегда, мы создали нашу референсную модель, которая слева на изображении. Мы повернули вторую модель сначала вокруг оси y, и затем вокруг оси z. Вращение использует в качестве пространства по-умолчанию локальное пространство. Это подразумевает, что после того, как мы повернули первую модель на 90 градусов вокруг оси y, ориентация оси z изменилась. Вторая модель использовала мировую систему координат, и там ориентация оси z осталась той же, даже когда мы повернули узел сцены.



Модель под номером 1 — это оригинальная система координат, которая у нас была. Под номером 2, мы видим систему координат после того, как мы повернули её на 90 градусов вокруг оси y. Под номером 3, мы вращали на 90 градусов вокруг оси z. Теперь давайте посмотрим на те же вращения, когда мы используем мировое пространство вместо локального пространства.



Здесь мы делаем те же вращения, но поскольку мы всегда использовали мировое пространство, мы не использовали изменённую систему координат, и, следовательно, мы получили другой результат.

Масштабирование в различных пространствах

Масштабирование всегда применяется к изначальной модели, так как нет различных пространств для масштабирования. Нет особого смысла масштабировать в различных пространствах, поскольку результат будет одинаковым.

Итог

Мы узнали много в этой главе о графе сцены, используемом Ogre 3D, и от том, как работать с ним, чтобы создавать сложные сцены.

Подробнее, мы разобрались со следующим:

- Что такое граф сцены, и как он работает
- Различные способы изменения позиции, вращения, и масштабирования узлов сцены
- Различные типы пространства, которые у нас есть для вращений и перемещений
- Как можно искусно использовать свойства графа сцены для создания сложных изображений

После изучения создания сложных сцен, в следующей главе, мы собираемся добавить свет, тени и создать нашу собственную камеру.

3 Камера, свет и тень

Мы уже узнали, как создавать сложную сцену, но без света и тени сцена не будет полной.

В этой главе мы узнаем о:

- Типах различных источников света, поддерживаемых Ogre 3D, и о том, как они работают
- Добавлении теней на сцену и различные доступные методы затенения
- Что такое камера и порт просмотра, и почему они нам нужны

Создание плоскости

Прежде, чем мы сможем добавить освещение к нашей сцене, нам нужно сначала добавить плоскость, на которую тени и свет будут проецироваться, и, следовательно, станут видимыми нам. Нормальному приложению плоскость не нужна, поскольку там будет местность или пол, чтобы спроецировать на него свет. Вычисление освещения должно работать без плоскости, но мы не сможем увидеть эффект от света.

Время для действия — создание плоскости

До сих пор, мы всегда загружали 3D-модели из файла. Теперь мы создадим одну непосредственно:

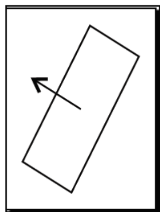
1. Удалите весь код в функции *createScene()*.
2. Добавьте следующую строку, чтобы определить плоскость в функции *createScene()*:
```
Ogre::Plane plane(Vector3::UNIT_Y, -10);
```
3. Теперь создайте плоскость в памяти компьютера:
```
Ogre::MeshManager::getSingleton().createPlane("plane",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
1500,1500,20,20,true,1,5,5,Vector3::UNIT_Z);
```
4. Создайте экземпляр плоскости:
```
Ogre::Entity* ent = mSceneMgr->createEntity("LightPlaneEntity", "plane");
```
5. Подключите плоскость к сцене:
```
mSceneMgr->getRootSceneNode()->createChildSceneNode()- >attachObject(ent);
```
6. Для того, чтобы получить что-нибудь, кроме белой плоскости, установите материал плоскости в один из существующих материалов:
```
ent->setMaterialName("Examples/BeachStones");
```
7. Скомпилируйте приложение и запустите это. Вы должны увидеть несколько темных камней.

Мы инвертировали цвета для удобства чтения!

Что произошло?

Мы только что создали плоскость и добавили её к сцене. На шаге 2 создали экземпляр класса *Ogre::Plane*. Этот класс описывает плоскость, используя вектор нормали к плоскости и смещение от нулевой точки, с использованием вектора нормали.

Вектор нормали (или короче говоря, просто нормаль), — часто используемая конструкция в 3D-графике. Нормаль поверхности — вектор, который стоит перпендикулярно к этой поверхности. Длина нормали обычно равна 1, и она широко используется в машинной графике для вычисления освещения и occlusion.



На шаге 3, мы использовали определение плоскости, чтобы создать из неё меш. Для этого, мы использовали объект Ogre 3D с именем *MeshManager*. Этот менеджер управляет мешами, которые не должны быть сюрпризом. Кроме управления мешами, которые мы загружаем из файла, он может также создать плоскость по нашему определению, а также много других вещей.

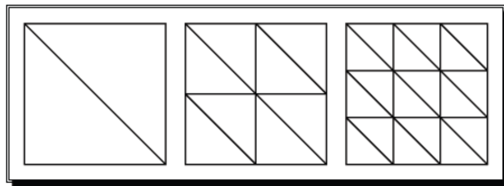
```
Ogre::MeshManager::getSingleton().createPlane("plane",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
1500,1500,20,20,true,1,5,5,Vector3::UNIT_Z);
```

Кроме определения плоскости, нам нужно дать ей имя. При загрузке мешей с диска, имя файла используется в качестве имени ресурса. Оно также должно принадлежать группе ресурсов, группы ресурса — похожи на пространства имён (namespaces) в C++. Третий параметр является определением плоскости, а четвертый и пятый параметры являются размером плоскости. Шестой и седьмой параметры используются, чтобы сказать сколько сегментов должна иметь плоскость. Чтобы понять, что значит сегмент, мы допустим небольшое отклонение, чтобы показать, как 3D-модели представлены в 3D-пространстве.

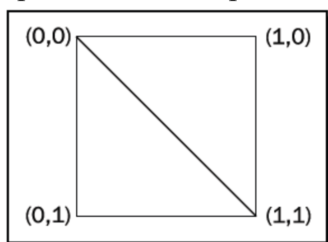
Представление модели в 3D

Чтобы отрендерить 3D-модель, она должна быть описана каким-то способом, чтобы компьютер мог понять и визуализировать её наиболее эффективно. Наиболее распространённая форма представления 3D-модели в приложении реального времени — треугольники. Наша плоскость может быть представлена с использованием двух

треугольников, чтобы сформировать четырёхугольник. С опцией сегментов для x- и y-осей, мы можем управлять тем, сколько треугольников будет сгенерировано для плоскости. На следующей картинке, мы видим треугольники, которые создают плоскость с одним, двумя, или тремя сегментами для каждой оси. Чтобы увидеть этот эффект, мы запускаем приложение, а затем нажимаем клавишу R. Это переключит первоначальный режим рендера на каркасный, где мы увидим треугольники. Другое нажатие клавиши изменит режим на точечный, где мы увидим только вершины треугольников. Следующее нажатие вернёт нормальный режим рендера.



После того, как мы определили, сколько сегментов нам нужно, мы передаем Логический (Boolean) параметр, который сообщает Ogre 3D, что мы хотим нормаль плоскости, которую нужно вычислить. Как сказано ранее, нормаль — это вектор, который стоит вертикально на поверхности плоскости. Следующие три параметра используются для текстурирования. Чтобы затекстурировать все точки чего-то, нужны текстурные координаты. Текстурные координаты сообщают движку рендера, как отобразить текстуру в треугольнике. Поскольку изображение является 2D-поверхностью, текстурные координаты также состоят из двух величин, а именно: x и y. Они представлены как кортеж (x,y). Диапазон величин текстурных координат нормирован от нуля до единицы. (0,0) означает верхний левый угол текстуры, а — (1,1) нижний правый угол. Иногда значения могут быть больше, чем 1. Это означает, что текстура может быть повторена в зависимости от способа набора. Эта тема будет шире объяснена в последующей главе. (2,2) могло бы повторить текстуру дважды по обеим осям. Десятый и одиннадцатый параметры сообщают Ogre 3D, как часто мы хотим, чтобы текстура была размножена по плоскости. Девятый параметр определяет, сколько текстурных координат нам нужно. Это может быть полезным при работе более чем с одной текстурой для одной поверхности. Последний параметр определяет направление «вверх» для наших текстур. Он также влияет на то, как текстурные координаты будут сгенерированы. Мы просто говорим, что ось z будет направлением «вверх» для нашей плоскости.



На 4 шаге мы создали экземпляр плоскости, которую мы только что сгенерировали с помощью MeshManager. Чтобы сделать это, нам нужно использовать имя, которое мы дали плоскости во время создания. Шаг 5 прикрепляет сущность к сцене.

На этапе 6 мы установили новый материал для экземпляра сущности. Каждая сущность имеет материал, назначенный ей. Материал описывает, какую текстуру использовать, как освещение взаимодействует с материалом, и ещё много чего. Мы узнаем обо всём этом в главе о материалах. Плоскость, которую мы создали, пока не имеет материала, и, следовательно, должна рендериться белым цветом. Поскольку мы хотим видеть эффект от освещения, которое мы создадим далее, белый не является наилучшим цветом для использования. Мы использовали материал, который уже определен в каталоге *media*. Этот материал просто добавляет текстуру с камнями к нашей плоскости.

Добавление точечного источника света

Теперь, когда мы создали плоскость, чтобы видеть эффект того, как свет воздействует на нашу сцену, нам нужно добавить источник света, чтобы что-то увидеть.

Время для действия — добавление точечного источника света

Мы создадим точечный свет и добавим его к нашей сцене, чтобы увидеть эффект, который он даёт нашей сцене:

1. Добавьте следующий код после установки материала для плоскости:
```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
```
2. Создайте источник света с именем *Light1* и сообщите Ogre 3D, что это — точечный источник света:
```
Ogre::Light* light1 = mSceneMgr->createLight("Light1");
light1->setType(Ogre::Light::LT_POINT);
```
3. Установите цвет освещения и позицию:
```
light1->setPosition(0,20,0);
light1->setDiffuseColour(1.0f,1.0f,1.0f);
```
4. Создайте сферу и установите её в позиции источника света, чтобы мы могли видеть, где он:
```
Ogre::Entity* LightEnt = mSceneMgr->createEntity("MyEntity",
"sphere.mesh");
Ogre::SceneNode* node3 = node->createChildSceneNode("node3");
node3->setScale(0.1f,0.1f,0.1f);
node3->setPosition(0,20,0);
node3->attachObject(LightEnt);
```
5. Скомпилируйте и запустите приложение; Вы должны увидеть каменную текстуру, освещённую белым светом, и белую сферу немного выше плоскости.

Что случилось?

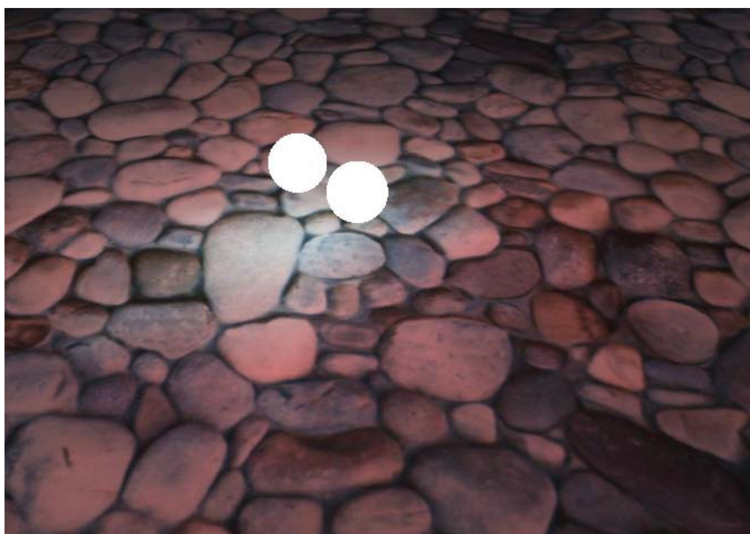
Мы добавили точечный источник света к нашей сцене, и использовали белую сферу для обозначения позиции источника.

На этапе 1, мы создали узел сцены и добавили его к корневому узлу сцены. Мы создали узел сцены, поскольку нам понадобится это позже, чтобы подключать светлую сферу. Первая интересная вещь происходит на этапе 2. Там мы создали новый источник света, используя менеджер сцены. Каждый источник света должен иметь уникальное имя, если мы решаем дать ему это имя. Если мы решаем не использовать имя, тогда Ogre 3D сгенерирует его за нас. Мы использовали в качестве имени *Light1*. После создания мы сообщили Ogre 3D, что

мы хотим создать точечное освещение. Есть три различных типа источников света, которые мы можем создать, а именно: точечные источники, прожекторы, и направленное освещение. Здесь мы создали точечное освещение; скоро мы создадим другие типы источников света. О точечном источнике света можно думать как о лампочке. Это — точка в пространстве, которая освещает все вокруг себя. На этапе 3 мы для созданного источника света установили его позицию и цвет. Цвет каждого источника света описывается кортежем (r,g,b). Все три параметра имеют диапазон от 0.0 до 1.0 и представляют соответствующую часть цвета. 'r' устанавливает красный, 'g' — зеленый, и 'b' — синий. (1.0,1.0,1.0) — это белый, (1.0,0.0,0.0) — красный, и так далее. Функция, которую мы вызвали, была *setDiffuseColour(r,g,b)*, которая принимает точно эти три параметра для цвета. Шаг 4 добавляет белую сферу в позицию источника света, чтобы мы могли видеть, где этот источник находится в сцене.

Have a go hero — добавляем второй точечный источник света

Добавьте второй источник света в позицию (20,20,20), который осветит сцену красным светом. Также добавьте другую сферу, чтобы показать, где находится этот источник. Вот как это должно выглядеть:



Добавление прожектора

Мы создали точечный источник света, а теперь мы создадим прожектор (spotlight) — второй тип источника света, который мы можем использовать.

Время для действия — добавление прожектора

Мы используем код, который мы создали прежде, и модифицируем его немного, чтобы увидеть как работает прожектор:

1. Удалите код, где мы создали источник света, и вставьте следующий код для создания нового узла сцены. Будьте осторожны, чтобы не удалить часть кода, который мы используем для создания *LigthEnt*. Затем добавьте следующий код:
```
Ogre::SceneNode* node2 = node->createChildSceneNode("node2");
node2->setPosition(0,100,0);
```
2. Снова, создайте источник света, но теперь установите его тип в прожектор (spotlight):
```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_SPOTLIGHT);
```
3. Теперь задайте некоторые параметры; мы обсудим, что они означают, позже:
```
light->setDirection(Ogre::Vector3(1,-1,0));
light->setSpotlightInnerAngle(Ogre::Degree(5.0f));
light->setSpotlightOuterAngle(Ogre::Degree(45.0f));
```

```
light->setSpotlightFalloff(0.0f);
```

4. Установите цвет освещения, и добавьте источник света к вновь созданному узлу сцены:
```
light->setDiffuseColour(Ogre::ColourValue(0.0f,1.0f,0.0f));
node2->attachObject(light);
```
5. Скомпилируйте и запустите приложение; это должно выглядеть примерно так:

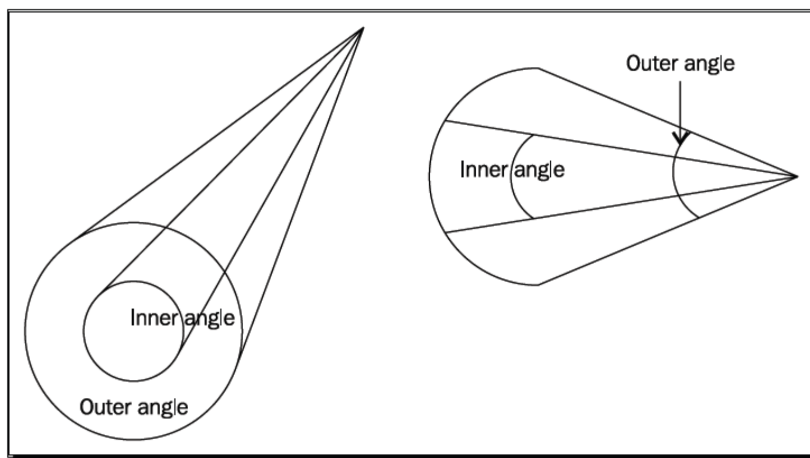
Что случилось?

Мы создали прожектор так же, как мы создали точечный источник света; мы только использовали несколько другие параметры для света.

Шаг 1 создаёт другой узел сцены, который понадобится позже. Шаг 2 создаёт свет так же, как мы делали прежде; мы только использовали другой тип источника света — на этот раз *Ogre::Light::LT_SPOTLIGHT*, чтобы получить прожектор. Шаг 3 — действительно интересен; там мы настроили различные параметры для прожектора.

Прожекторы

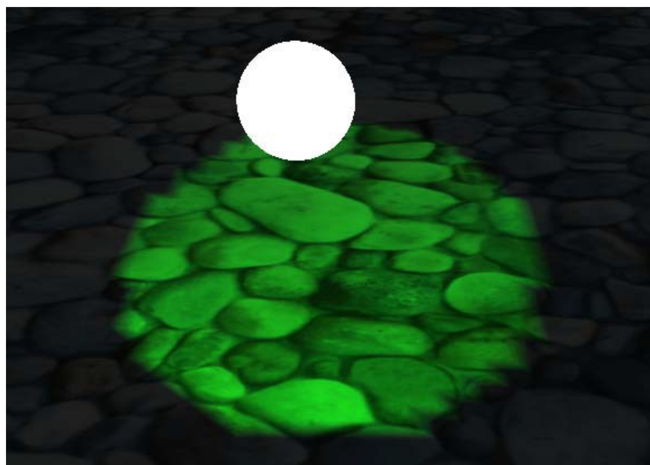
Прожекторы — просто подобны фонарям по своему эффекту. У них есть позиция, где они находятся, и направление, в котором они освещают сцену. Это направление было первой вещью, которую мы установили после создания света. Направление (direction) просто определяет, куда указывает прожектор. Следующими двумя параметрами мы установили внутренний и внешний углы прожектора. Внутренняя часть прожектора освещает область с полной мощностью исходного источника света. Внешняя часть конуса использует меньшую мощность света, освещая объекты. Это сделано, чтобы эмулировать эффекты реального фонаря. Реальный фонарь также имеет внутреннюю часть и внешнюю, которая освещает область слабее, чем центр прожектора. *InnerAngle* и *OuterAngle*, которые мы установили, определяют, насколько большими внутренняя и внешняя части должны быть. После установки углов, мы установили параметр спада (falloff). Этот параметр описывает, сколько мощности свет теряет, освещая внешнюю часть светового конуса. Чем дальше освещаемая точка находится от внутреннего конуса, тем сильнее спад влияет на точку. Если точка расположена за пределами внешнего конуса, тогда она не освещена прожектором.



Мы установили *falloff* (спад) в нуль. Теоретически, мы должны видеть отличный световой круг на плоскости, но он получился довольно смазанным и деформированным. Причина этого в том, что освещение, которое мы используем в данное время, использует точки треугольника плоскости, чтобы вычислять и применять освещение. При создании плоскости, мы сообщили Ogre 3D, что плоскость должна быть создана из 20X20 сегментов. Это — довольно низкое разрешение для такой большой плоскости, что означает, что свет не может вычисляться достаточно точно, поскольку здесь слишком мало точек для применения к области, чтобы сделать круг плавным. Так, чтобы получить лучшее качество рендера, мы должны увеличить количество сегментов плоскости. Скажем, мы увеличим количество сегментов с 20 до 200. Код создания плоскости станет похожим на такой после увеличения:

```
Ogre::MeshManager::getSingleton().createPlane("plane",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
1500,1500,200,200,true,1,5,5,Vector3::UNIT_Z);
```

Теперь после перекомпиляции и перезапуска приложения, мы получим хорошее круглое пятно света от нашего прожектора.



Круг все еще не идеальный; если нужно, мы могли бы увеличить количество сегментов плоскости, чтобы сделать его совершенным. Существуют различные методы освещения, которые дают лучшие результаты для плоскости с низким разрешением, но они — довольно сложны, и сейчас только внесут путаницу. Но даже при сложных методах освещения, основы будут теми же, и мы сможем изменить схему нашего освещения позже, используя тот же источник света, что мы создали здесь.

На этапе 4 мы видим другой способ описания цвета в Ogre 3D. Прежде мы использовали три величины, (r,g,b), чтобы задать диффузный цвет нашего источника света. Здесь мы использовали *Ogre::ColourValue (r,g,b)*, который в основе своей то же самое, но изолирован (инкапсулирован) в класс с несколькими дополнительными функциями, и, таким

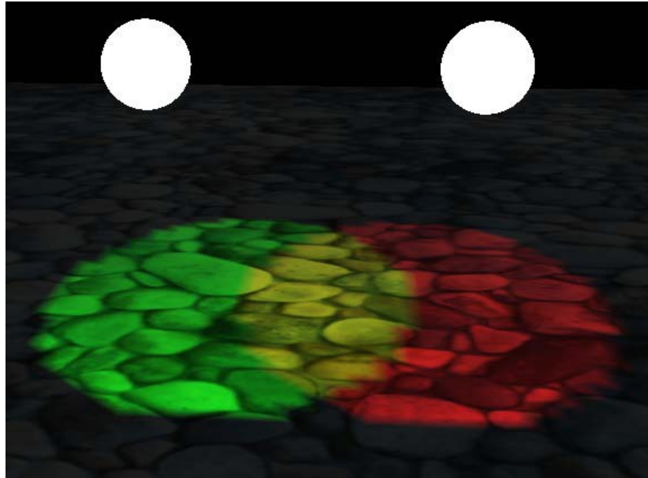
образом, он делает смысл этого параметра более ясным.

Популярная викторина — различные источники света

Опишите различия между точечным светильником и прожектором в нескольких словах.

Have a go hero — смешивание цветов освещения

Создайте второй прожектор, расположенный в другом месте по сравнению с первым прожектором. Дайте этому прожектору красный цвет и спозиционируйте его таким образом, чтобы круги обоих прожекторов слегка перекрывали друг друга. Вы должны увидеть, как цвет смешивается в области, где зеленый и красный лучи перекрываются.



Направленное освещение

Мы создали прожекторы и точечное освещение. Теперь мы собираемся создать последний тип освещения — направленное (directional) освещение. Направленный свет — это свет от источника, находящегося далеко, и он имеет только направление и цвет, но нет никакого светового конуса или радиуса, подобно освещению прожекторов или точечных светильников. О нём можно думать, как о солнце. Для нас солнечный свет исходит с одного направления, направления на солнце.

Время для действия — создание направленного освещения

1. Удалите весь старый код в функции *createScene()*, за исключением кода, связанного с плоскостью.
2. Создайте источник света, и задайте его тип на направленный (directional) свет:
```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
```
3. Задайте освещению белый цвет и направление лучей вниз-вправо:
```
light->setDiffuseColour(Ogre::ColourValue(1.0f,1.0f,1.0f));
light->setDirection(Ogre::Vector3(1,-1,0));
```
4. Скомпилируйте и запустите приложение.

Что произошло?

Мы создали направленное освещение и настроили, чтобы оно светило вниз и вправо с помощью *setDirection(1,-1,0)*. В предыдущих примерах у нас всегда была относительно черная плоскость, а небольшая часть плоскости была освещена нашим точечным источником или прожектором. Здесь мы использовали направленный свет и, следовательно, осветили всю плоскость. Как было сказано раньше, о направленном свете можно думать, как о солнце, а солнце не имеет радиуса затухания или чего-нибудь ещё. Так, когда оно сияет, оно освещает всё вокруг; то же является истиной для нашего направленного освещения.

Популярная викторина — различные типы освещения

Вспомните все три типа источника света, которые есть в Ogre 3D, и укажите различия.

Отсутствующая вещь

Мы уже добавили свет к нашей сцене, но что-то пропустили. Что именно пропущено, будет показано в следующем примере.

Время для действия — поиск пропущенного

Мы используем предложенный ранее код, чтобы найти, что пропущено в нашей сцене.

1. После создания освещения, добавьте код создания экземпляра *Sinbad.mesh*, а также создайте узел и подключите модель к нему:
```
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad", "Sinbad. mesh");
Ogre::SceneNode* SinbadNode = node- >createChildSceneNode("SinbadNode");
```
2. Затем увеличьте размер Синбада в три раза, и переместите его немного вверх; в противном случае, он будет воткнут в плоскость. Также добавьте его к узлу сцены, чтобы он был отрендерен:
```
SinbadNode->setScale(3.0f,3.0f,3.0f);
SinbadNode->setPosition(Ogre::Vector3(0.0f,4.0f,0.0f));
SinbadNode->attachObject(Sinbad);
```
3. Скомпилируйте и запустите приложение.

Что произошло?

Мы добавили экземпляр Синбада в нашу сцену. Наша сцена все еще освещена, но мы видим, что Синбад не отбрасывает тень, что довольно нереалистично. Следующим шагом нужно добавить тени в нашу сцену.

Добавление теней

Без теней 3D-сцена не выглядит полностью завершенной, так давайте добавим их.

Время для действия — добавление тени

Используйте предыдущий код.

1. После добавления всего остального кода в функции *createScene()*, добавьте следующую строку:
```
mSceneMgr->setShadowTechnique(Ogre:: SHADOWTYPE_STENCIL_ADDITIVE);
```
2. Скомпилируйте и запустите приложение.

Что произошло?

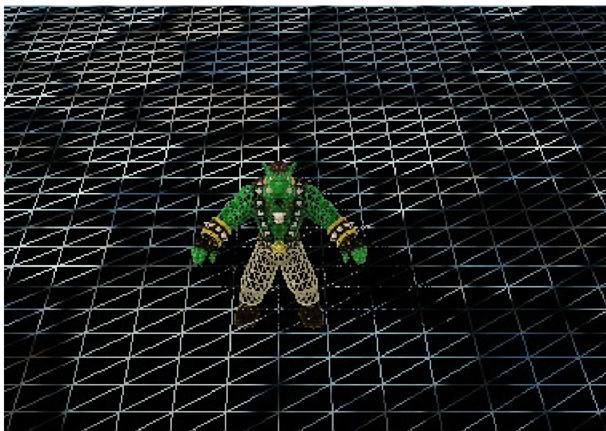
С помощью всего одной строки мы добавили тени в нашу сцену. Ogre 3D делает остальную часть работы за нас. Ogre 3D поддерживает различные методы затенения. Мы использовали добавленные (additive) стенсильные (stencil — трафарет) тени. Трафарет означает специальный буфер текстуры, использованный при рендере сцены. Добавленные подразумевает, что сцена рендерится один раз из перспективы камеры, и вклад каждого источника света будет просуммирован в окончательном изображении. Эта техника создает хорошие результаты, но также очень дорога, поскольку каждый источник света добавляет дополнительный проход рендера. Мы не будем вдаваться в подробности того, как работают тени, поскольку это — сложная область. Много книг можно написать об этой теме, и кроме того, методы затенения быстро меняются и интенсивно исследуются. Если Вас заинтересовала эта тема, Вы можете найти интересные статьи в книжной серии GPU Gems корпорации NVIDIA, книжной серии ShaderX, и в протоколах конференции Siggraph (http://www.siggraph.org/).

Создание камеры

До сих пор мы всегда использовали камеру, которую создавал за нас класс *ExampleApplication*. Теперь давайте создадим её сами. Камера, как и предполагает название, снимает часть нашей сцены из определенной позиции. В конкретный момент времени может только быть одна активная камера, поскольку у нас только есть один вывод, то есть, наш монитор. Но вполне возможно использовать несколько камер на одной сцене, когда каждая рендерит после другой.

Время для действия — создание камеры

На этот раз мы не будем переделывать функцию *createScene()*; просто оставьте её такой, какой она была — с экземпляром Синбада и тенью:

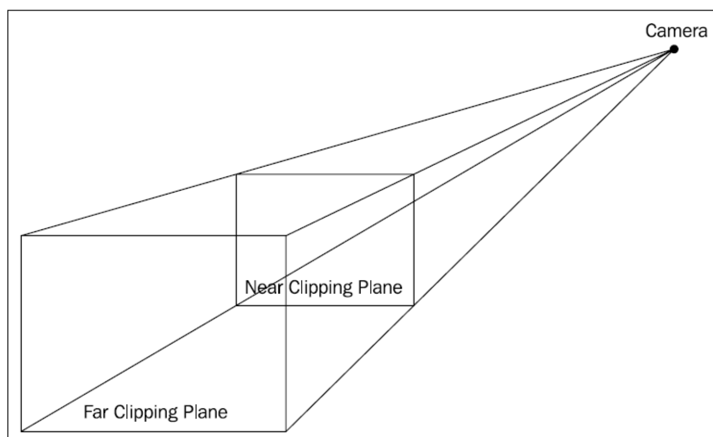
1. Создайте новую пустую функцию с именем *createCamera()* в классе *ExampleApplication*:
```
void createCamera() {
}
```
2. Создайте новую камеру, с именем *MyCamera1* и присвойте её переменной mCamera:
```
mCamera = mSceneMgr->createCamera("MyCamera1");
```
3. Задайте позицию камеры и направьте её взгляд на нулевую точку:
```
mCamera->setPosition(0,100,200);
mCamera->lookAt(0,0,0);
mCamera->setNearClipDistance(5);
```
4. Теперь измените режим рендера в каркасный:
```
mCamera->setPolygonMode(Ogre::PM_WIREFRAME);
```


5. Скомпилируйте и запустите приложение.

Что произошло?

Мы переписали метод *createCamera()*, который изначально создавал камеру и ставил её в определённое место. После создания, мы задали позицию, и использовали функцию *lookat()*, чтобы настроить взгляд камеры на начало координат. Следующим шагом мы установили ближнее (*near*) расстояние отсечения. Камера может видеть только части 3D-сцены, так что рендерить её полностью было бы пустой тратой драгоценного времени графического и центрального процессоров. Для того, чтобы предотвратить это, перед рендером большие части сцены «отрезаются» от неё Менеджером сцены. Рендерятся только объекты, видимые в камере. Этот шаг называется **culling** (выбраковка). Только те объекты, которые находятся дальше, чем плоскость отсечения *near*, и ближе, чем плоскость отсечения *far*, будут отрендерены, т.е. только когда они находятся в усечённой пирамиде, называемой view frustum камеры. Вид *frustum* является пирамидой с отрезанным верхом; только объекты, находящиеся вней, камера может увидеть. Больше информации можно найти по адресу

http://www.lighthouse3d.com/opengl/viewfrustum/. Мы установили плоскость отсечения *near* в значение 5. Когда Вы используете большее значение, то большая часть сцены, расположенная около камеры, это будет отрезана и станет невидимой.



Затем мы изменили режим рендера на каркасный. Этот эффект мы получаем при нажатии клавиши **R**, как сказано ранее, так же, как тогда, когда мы захотели увидеть треугольники плоскости. С помощью **R** этот эффект также можно отменить. Когда приложение стартует, мы видим различие по сравнению с тем, что было раньше; камера — теперь выше Синбада и направлена вниз под ним. Перед перезаписью функции *createCamera()*, при начальном положении камера зависала невысоко над плоскостью, взгляд направлен на начало координат. С помощью *setPosition(0,100,200)* мы установили камеру выше, чем прежде; следующий скриншот показывает изменение. Один интересный аспект, который мы можем наблюдать, то, что даже после того, как мы создали наш собственный экземпляр камеры, мы все еще можем двигаться по сцене, как прежде. Это возможно, поскольку мы использовали переменную-член *mCamera* класса *ExampleApplication*. Она используется в *ExampleApplication* для управления нашей камерой, и, таким образом, может быть модифицирована. Одна важная характеристика камеры — она может также быть приложена к узлу сцены, и будет реагировать тем же способом, которым сущность (entity) действует, когда она приложена к узлу сцены.

Have a go hero — делаем больше

Попробуйте использовать различные позиции и точки обзора (look at) , чтобы увидеть, как это влияет на стартовую позицию камеры.

Также попробуйте увеличить расстояние отсечения *near*, и посмотрите, какой эффект это возымеет. Это должно приводить к странным изображениям, подобно следующему, где мы можем посмотреть в голову Синбада. Расстояние отсечения *near* было установлено на 50, чтобы получить эту картинку.

Создание порта просмотра

Тесно переплетено с понятием камеры другое понятие — порт просмотра (**viewport**). Поэтому мы создадим наш собственный порт просмотра. Порт просмотра — 2D-поверхность, которая используется для рендера. Мы можем думать о нём, как о бумаге на которой получается фото. Бумага имеет цвет фона (background color), и если фото не закрывает этот регион, то цвет фона будет видимым.

Время для действия — делаем что-то, что иллюстрирует вещь "в действии"

Мы используем код, который мы использовали прежде, и снова создим новый пустой метод:

1. Удалите вызов функции *setShadowTechnique()* из функции *createCamera()*. Мы не хотим, чтобы наша сцена была изображена каркасном режиме.
2. Создайте пустой метод *createViewports()*:
```
void createViewports() {
}
```
3. Создайте порт просмотра:
```
Ogre::Viewport* vp = mWindow->addViewport(mCamera);
```
4. Установите цвет фона и соотношение ширины к высоте:
```
vp->setBackgroundColour(ColourValue(0.0f,0.0f,1.0f));
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
```
5. Скомпилируйте и запустите приложение.

Что произошло?

Мы создали порт просмотра. Чтобы сделать это, нам нужно передать камеру в функцию. Каждый порт просмотра может рендерить вид только одной камеры, так что Ogre 3D обеспечивает одна камера, данную при создании. Конечно, камеру можно поменять впоследствии, используя подходящие функции *getter* и *setter*. Наиболее заметное изменение в том, что цвет фона изменился с черного на синий. Причина должна быть очевидна: новый порт просмотра имеет синий цвет фона; мы задали это на шаге 3. Также на этом шаге мы установили **aspect ratio** — коэффициент соотношения ширины к высоте визуализируемого изображения; в математических терминах: aspect ratio = ширина окна, разделённая на высоту окна.

Have a go hero — играем с различными соотношениями

Попробуйте поиграть с разными значениями aspect ratio и посмотрите, как это влияет на полученное изображение. Также измените цвет фона. Вот картинка, где ширина в соотношении разделена на пять.

Итог

В этой главе мы добавили освещение и тени к нашей сцене, создали порт просмотра, и работали с *viewfrustum*.

Подробнее, мы изучили

- Какие могут быть источники света, и как они могут модифицировать вид нашей сцены
- Добавление теней на нашу сцену
- Создание нашей собственной камеры, frustum, и порта просмотра:

В следующей главе мы узнаем как обрабатывать ввод пользователя с клавиатуры и мыши. Мы также узнаем, что такое *FrameListener*, и как использовать его.

4 Получение ввода пользователя и использование Frame Listener

До сих пор мы всегда создавали изображения, которые были статическими, и в них ничего не двигалось. В этой главе мы изменим это.

В этой главе, мы будем:

- Изучать *FrameListener*
- Узнавать, как обрабатывать ввод пользователя
- Объединять оба понятия, чтобы создать наше собственное управление камерой

Итак, давайте продолжим…

Подготовка сцены

Перед добавлением движения на нашу сцену, нам нужна сама сцена, на которую можно что-то добавить. Так давайте создадим сцену.

Время для действия — подготовка сцены

Мы используем немного отличающуюся версию сцены из предшествующей главы:

1. Удалите весь код в функциях *createScene()* и *createCamera()*.
   Назовите класс, наследуемый от ExampleApplication именем Example25. — доб. пер.
2. Удалите функцию *createViewports()*.
3. Добавьте новую переменную-член к классу. Эта переменная-член является указателем на узел сцены:
```
private:
Ogre::SceneNode* _SinbadNode;
```
4. Создайте плоскость и добавьте её к сцене, используя метод *createScene()*:
```
Ogre::Plane plane(Vector3::UNIT_Y, -10);
Ogre::MeshManager::getSingleton().createPlane("plane",
      ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
      1500,1500,200,200,true,1,5,5,Vector3::UNIT_Z);
Ogre::Entity* ent = mSceneMgr->createEntity("LightPlaneEntity", "plane");
mSceneMgr->getRootSceneNode()->createChildSceneNode()- >attachObject(ent);
ent->setMaterialName("Examples/BeachStones");
```
5. Затем добавьте освещение к сцене:
```
Ogre::Light* light = mSceneMgr->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1,-1,0));
```
6. Нам также нужен экземпляр Синбада; создайте узел и подключите экземпляр к нему:
```
Ogre::SceneNode* node = mSceneMgr->createSceneNode("Node1");
mSceneMgr->getRootSceneNode()->addChild(node);
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad", "Sinbad. mesh");
_SinbadNode = node->createChildSceneNode("SinbadNode");
_SinbadNode->setScale(3.0f,3.0f,3.0f);
_SinbadNode->setPosition(Ogre::Vector3(0.0f,4.0f,0.0f));
_SinbadNode->attachObject(Sinbad);
```

7. Мы также хотим, чтобы в этой сцене отображались тени; так что активизируем их:
```
mSceneMgr->setShadowTechnique(SHADOWTYPE_STENCIL_ADDITIVE);
```
8. Создайте камеру, установите её в позицию (0,100,200), и задайте направление взгляда на (0,0,0); не забудьте добавить код к функции *createCamera()*:
```
mCamera = mSceneMgr->createCamera("MyCamera1");
mCamera->setPosition(0,100,200);
mCamera->lookAt(0,0,0);
mCamera->setNearClipDistance(5);
```
9. Скомпилируйте и запустите приложение, и Вы должны увидеть следующее изображение:

Что произошло?

Мы использовали знания, полученные в предшествующих главах, для создания сцены. Вы должны быть способны понимать, что происходит. Если нет, вы должны возвратиться к предшествующим главам и читать их снова, пока не поймёте всё.

Добавление движения на сцену

Мы создали нашу сцену; теперь давайте добавим движение к ней.

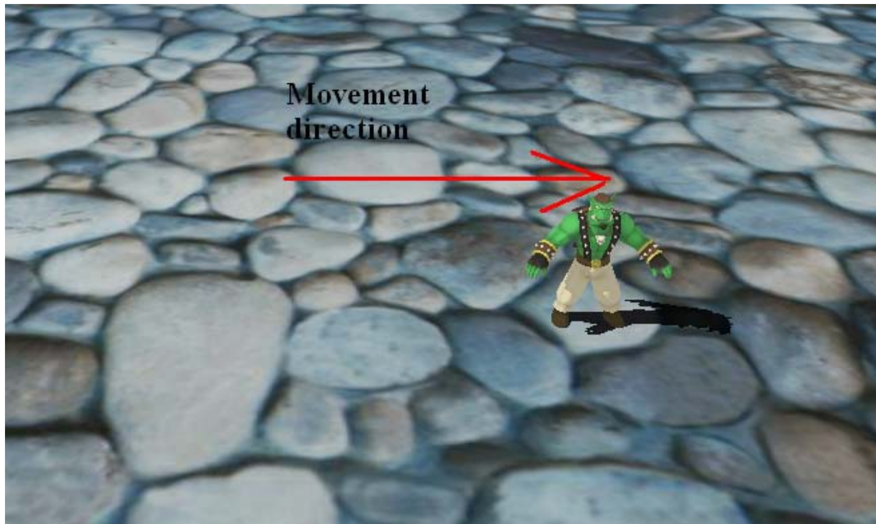
Время для действия — добавление движения к сцене

До сих пор у нас был только один класс, а именно, *ExampleApplication*. На этот раз нам нужен ещё один:

1. Создайте новый класс, назовите его Example25FrameListener, и сделайте его публично унаследованным от Ogre::FrameListener:
```
class Example25FrameListener : public Ogre::FrameListener
{
};
```
2. Добавьте приватную переменную-член, с типом указатель на *Ogre::SceneNode*, и именем *_node*:
```
private:
      Ogre::SceneNode* _node;
```
3. Добавьте публичный конструктор, который принимает указатель на *Ogre::SceneNode* в качестве параметра и присваивает его переменной-члену:
```
public:
      Example25FrameListener(Ogre::SceneNode* node)
      {
```

```
        _node = node;
    }
```

4. Добавьте новую функцию, с именем *frameStarted(FrameEvent& evt)*, который сдвигает (translate) узел на (0.1,0,0), а затем возвращает истину (*true*):
```
bool frameStarted(const Ogre::FrameEvent &evt)
{
    _node->translate(Ogre::Vector3(0.1,0,0));
    return true;
}
```
5. *Далее работаем с классом Example25 — доб. пер.*
Добавьте в секцию *private* новую переменную-член для сохранения указателя на *FrameListener*, который мы создадим позже:
```
Ogre::FrameListener* FrameListener;
```
6. Добавьте конструктор, с начальным присваиванием указателя в *NULL*, и деструктор, который уничтожает *FrameListener* при закрытии приложения:
```
Example25()
{
    FrameListener = NULL;
}
~Example25()
{
    if(FrameListener)
    {
    delete FrameListener;
    }
}
```
7. Теперь создайте новую функцию в *ExampleApplication* с именем *createFrameListener*. В этой функции создайте экземпляр *FrameListener*, который мы определили, и добавьте его, используя *mRoot*:
```
void createFrameListener()
{
    FrameListener = new Example25FrameListener(_SinbadNode);
    mRoot->addFrameListener(FrameListener);
}
```
8. Скомпилируйте и запустите приложение. Вы должны увидеть то же изображение, как видели раньше, но на этот раз, Синбад двигается вправо, но Вы не можете перемещать камеру или закрывать приложение с клавишей *Escape*. Для того, чтобы закрыть приложение, щелкните кнопку X в консоли окна, или, если Вы запустили приложение с консоли, Вы можете использовать *CTRL+C*.



Что произошло?

Мы добавили новый класс к нашему коду, который перемещает наш узел сцены.

FrameListener

Новое понятие, с которым мы здесь столкнулись — понятие *FrameListener*. Как и предполагает имя, *FrameListener* основан на паттерне (шаблоне) наблюдателя (observer). Мы можем добавить экземпляр класса, который наследуется от интерфейса *Ogre::FrameListener* в наш корневой экземпляр Ogre 3D, используя метод *addFrameListener()* класса *Ogre::Root*. Когда этот экземпляр класса добавлен, наш класс извещается, если случаются определенные события. В этом примере, мы переписали метод *frameStarted()*. Перед тем, как кадр (под кадром, мы обозначаем единичное изображение сцены) будет отрендерен, *Ogre::Root* перебирает все добавленные к нему экземпляры *FrameListener*, и вызывает у каждого метод *frameStarted()*. В нашей реализации (смотри шаг 4) этой функции, мы перемещаем узел на 0.1 единицу по оси x. Этот узел был передан в *Framelistener* в его конструкторе. Следовательно, всякий раз, когда сцена рендерится, узел смещается немного, и, в результате, перемещает модель.

Как мы видели во время работы приложения, мы не можем перемещать нашу камеру или выйти из нашего приложения используя клавишу *Escape*. Дело в том, что всё это действовало посредством *FrameListener*, который поставлялся с каркасом *ExampleApplication*. Каркас *ExampleApplication* поставляется с SDK. Теперь, когда мы заменили его нашей собственной реализацией, мы больше не можем использовать функции предлагаемого *FrameListener*. Но мы повторно реализуем большинство из них в этой главе, так что никаких беспокойств. Если нужно, мы все еще можем вызывать функции базового класса, чтобы получить поведение по-умолчанию.

На этапе 4, наша функция возвращает *истину (true)*. Если возвратить *ложь (false)*, Ogre 3D должен интерпретировать это как сигнал отключить цикл рендера, и, с помощью этого, приложение должно быть закрыто. Мы используем этот факт в повторной реализации функции "press Escape to exit the application" (нажмите *Escape*, чтобы выйти из приложения).

Популярная викторина — шаблон проектирования FrameListener

На каком шаблоне (паттерне) проектирования основана концепция *FrameListener*?

a. Decorator

b. Bridge

c. Observer

Модификация кода, чтобы движение зависело от времени, а не от кадра

В зависимости от вашего компьютера, модель на сцене может перемещаться слишком быстро, или слишком медленно, или с нормальной скоростью. Причина различных скоростей перемещения модели в том, что в нашем коде мы перемещаем модель на 0.1 единицу по оси x перед каждым новым кадром, который будет отредерен. Новый компьютер может оказаться способен рендерить сцену со 100 кадрами в секунду; это должно перемещать модель на 10 единиц в секунду. При использовании старого компьютера, у нас может оказаться 30 кадров в секунду, тогда модель должна перемещаться только на 3 единицы. Это только одна треть по сравнению с новыми компьютерами. В нормальной ситуации нам хотелось бы, чтобы наше приложение работало стабильно на различных платформах, и было способно работать на одинаковой скорости. В Ogre 3D этого легко достигнуть.

Время для действия — добавление перемещения, основанного на времени

Мы используем предыдущий код и модифицируем только одну строку:

1. Измените строку, где мы перемещаем (translate) узел, на такую:
```
_node->translate(Ogre::Vector3(10,0,0) * evt.timeSinceLastFrame);
```
2. Скомпилируйте и запустите приложение. Вы должны увидеть то же изображение, как и до этого, только модель может перемещаться на другой скорости.

Что произошло?

Мы изменили наше перемещение модели из основанного-на-кадрах в основанное-на-времени. Мы достигли этого, добавив простое умножение. Как сказано чуть раньше, кадровое перемещение имеет некоторые недостатки. Перемещение, основанное на времени лучше, поскольку мы получаем одинаковое перемещение во всех компьютерах и имеем больше контроля над скоростью перемещения. На этапе 1 мы использовали то, что Ogre 3D передает *FrameEvent* при вызове метода *frameStarted()*. Этот класс содержит время в секундах, прошедшее после последнего отрендеренного кадра:

Эта строка использует эти данные, чтобы вычислять, на сколько мы хотим переместить нашу модель в этом кадре. Мы используем вектор и умножаем его на время в секундах, прошедшее после предыдущего кадра. В нашем случае, мы используем вектор (10,0,0). Это означает, что мы хотим, чтобы наша модель перемещалась на 10 единиц по оси x в секунду. Скажем, мы рендерим с 10 кадрами в секунду; тогда для каждого кадра, *evt.timeSinceLastFrame* будет равно 0.1f. В каждом кадре мы умножаем *evt.timeSinceLastFrame* на вектор (10,0,0), что в результате даст вектор (1,0,0). Этот результат применяется к узлу сцены в каждом кадре. С 10 кадрами в секунду это даст перемещение на (10,0,0) в секунду, что в точности является нужной нам величиной скорости перемещения модели.

Популярная викторина — различие между перемещением, основанным на времени и на кадрах

Опишите своими словами различия между перемещением, основанным на кадрах, и

основанным на времени.

Have a go hero — добавление второй модели

Добавьте вторую модель к сцене, и пусть она перемещается в противоположном к первой модели направлении.

Добавление поддержки ввода

У нас теперь есть движущаяся сцена, но мы хотели бы иметь возможность выходить из нашего приложения, как раньше. Следовательно, теперь мы собираемся добавить поддержку ввода, и при нажатии Escape мы выйдем из нашего приложения. До сих пор, мы использовали только Ogre 3D; теперь мы также будем использовать **OIS (Объектно-Ориентированная Система Ввода)**, которая поставляется вместе с Ogre 3D SDK, поскольку её использует *ExampleFrameListener*, но в остальном эта библиотека полностью независима от Ogre 3D.

Время для действия — добавление поддержки ввода

Опять, мы используем предыдущий код и добавим необходимое, чтобы получить поддержку ввода:

1. Нам нужно добавить новый параметр к конструктору нашего слушателя (listener). Нам нужен указатель на окно рендера, которое используется Ogre 3D. Чтобы добавить новый параметр, код должен выглядеть похожим на это:
```
Example27FrameListener(Ogre::SceneNode* node, RenderWindow* win)
```
2. При изменении конструктора нам также нужно изменить экземпляр:
```
Ogre::FrameListener* FrameListener = new Example27FrameListener(_SinbadNode,
mWindow);
```
3. После этого нам нужно добавить код в конструктор слушателя. Сначала нам нужны две вспомогательные переменные:
```
size_t windowHnd = 0;
std::stringstream windowHndStr;
```
4. Теперь спросим Ogre 3D хендл окна, в которое он рендерит:
```
win->getCustomAttribute("WINDOW", &windowHnd);
```
5. Конвертируем хендл в строку:
```
windowHndStr << windowHnd;
```
6. Создадим список параметров для OIS и добавим к нему хендл окна:
```
OIS::ParamList pl;
pl.insert(std::make_pair(std::string("WINDOW"), windowHndStr. str()));
```
7. Создадим систему ввода, используя этот список параметров:
```
_man = OIS::InputManager::createInputSystem( pl );
```
8. Затем создадим клавиатуру, чтобы мы могли проверять наличие ввода пользователя:
```
_key = static_cast<OIS::Keyboard*> (_man->createInputObject(OIS::OISKeyboard,
false ));
```
9. Всё, что мы создаём, мы должны уничтожить. Добавьте деструктор к *FrameListener*, который уничтожает наши созданные объекты OIS:
```
~Example27FrameListener()
{
       _man->destroyInputObject(_key);
       OIS::InputManager::destroyInputSystem(_man);
}
```
10. После того, как мы завершили инициализацию, добавьте следующий код в метод

frameStarted() после перемещения узла и перед выходом:

```
_key->capture();
if(_key->isKeyDown(OIS::KC_ESCAPE))
{
      return false;
}
```

11. Добавьте используемые объекты ввода в качестве переменных-членов к *FrameListener*:

```
OIS::InputManager* _man;
OIS::Keyboard* _key;
```

12. Скомпилируйте и запустите приложение. Теперь Вы должны иметь возможность выходить из приложения, нажимая клавишу *Escape.*

Что произошло?

Мы создали экземпляр системы ввода и использовали его, чтобы захватывать нажатие клавиш пользователем. Поскольку для создания системы ввода нам нужен хендл окна, мы изменили конструктор *FrameListener*, чтобы он получил указатель на окно рендера. Это было сделано на этапе 1. Затем мы преобразовали хендл из числа в строку, и добавили эту строку в список параметров OIS. С этим списком мы создали наш экземпляр системы ввода.

Хендл окна

Хендл окна является просто числом, которое используется как идентификатор для определенного окна. Это число создаётся операционной системой, и каждое окно имеет уникальный хендл. Системе ввода нужен этот хендл, поскольку без него она не сможет получать события ввода. Ogre 3D создает окно для нас. Так что для получения хендла окна нам нужно опросить его в следующей строке:

```
win->getCustomAttribute("WINDOW", &windowHnd);
```

Существует несколько атрибутов, которые имеет окно рендера, так что Ogre 3D применяет общую getter-функцию. Она также должна быть платформо-независимой; каждая платформа имеет собственные типы переменных для хендлов окна, так что это единственный путь для кроссплатформенности. *WINDOW* является ключевым словом для хендла окна. Нам нужно передать в функцию указатель для сохранения значения хендла; в этот указатель будет записано его значение. После получения хендла мы преобразуем его в строку, используя *stringstream*, поскольку это то, что ожидает OIS. OIS имеет ту же проблему и использует то же решение. Во время создания, мы передаём OIS список с парами параметров, состоящими из строкового идентификатора и строки со значением. На этапе 6 мы создали этот список параметров и добавили в него строку хендла окна. Шаг 7 использует этот список для создания системы ввода. С системой ввода мы можем создать наш интерфейс клавиатуры на этапе 8. Этот интерфейс будет использоваться для опроса системы — какие клавиши нажаты пользователем? Это делает код шага 10. Каждый раз до того, как мы отрендерим кадр, мы захватываем новое состояние клавиатуры, используя функцию *capture()*. Если бы мы не вызывали эту функцию, мы не получили бы нового состояния клавиатуры, и, следовательно, мы никогда не получим никаких событий клавиш. После обновления состояния, мы запрашиваем клавиатуру, нажата ли клавиша escape прямо сейчас. Если это так, то мы знаем, что пользователь хочет покинуть приложение. Это означает, что мы должны возвратить ложь, чтобы дать понять Ogre 3D, что мы хотим отключить приложение. В противном случае, если пользователь хочет, чтобы приложение продолжало работать, мы возвращаем истину, чтобы продолжить выполнение приложения.

Популярная викторина — вопросы об окне

Что такое хендл окна и как он используется нашим приложением и операционной системой?

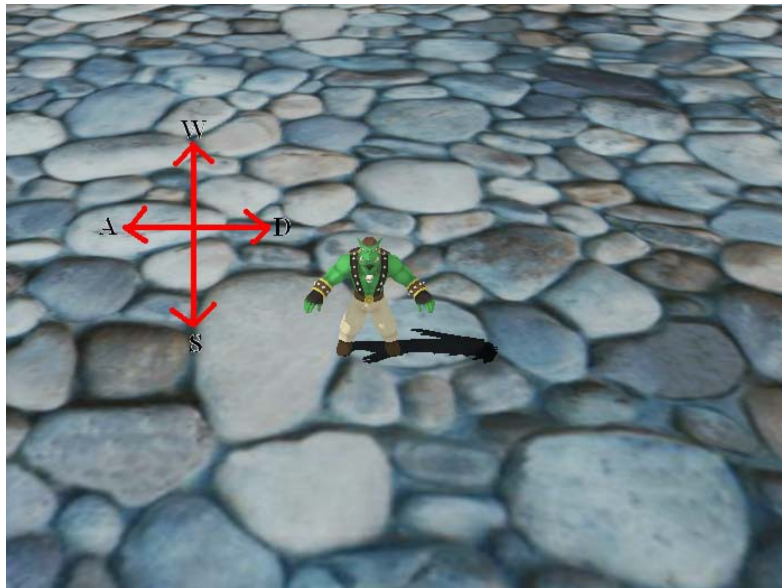
Добавление движения модели

Теперь, когда у нас есть возможность получать ввод пользователя, давайте используем его для управления перемещением Синбада по плоскости.

Время для действия — управление Синбадом

Мы используем предыдущую программу и добавим этот код там, где нам нужно, как мы делали прежде. Мы создадим управление клавишами WASD для Синбада следующим образом:

1. Замените строку, где мы перемещаем узел в *FrameListener* с нулевым вектором при вызове *translate*:
```
Ogre::Vector3 translate(0,0,0);
```
2. Затем добавьте следующий запрос клавиатуры после запроса escape:
```
if(_key->isKeyDown(OIS::KC_W))
{
      translate += Ogre::Vector3(0,0,-10);
}
```
3. Теперь добавьте код для трех остальных клавиш. Он в основном одинаковый, меняются только коды клавиш и направления векторов:
```
if(_key->isKeyDown(OIS::KC_S))
{
      translate += Ogre::Vector3(0,0,10);
}
if(_key->isKeyDown(OIS::KC_A))
{
      translate += Ogre::Vector3(-10,0,0);
}
if(_key->isKeyDown(OIS::KC_D))
{
      translate += Ogre::Vector3(10,0,0);
}
```
4. Теперь используйте вектор *translate*, чтобы переместить модель, и не забудьте, что используется перемещение, основанное на времени, а не на кадрах:
```
_node->translate(translate*evt.timeSinceLastFrame);
```
5. Скомпилируйте и запустите приложение, и Вы сможете управлять Синбадом с помощью клавиш WASD



Что произошло?

Мы добавили базовое управление перемещением, использующее клавиши WASD. Мы опросили все четыре клавиши и построили суммарный вектор перемещения. Затем мы применили этот вектор к модели, используя перемещение, основанное на времени.

Have a go hero — использование показателя скорости для перемещения

Недостаток этого метода в том, что когда мы захотим изменить скорость перемещения модели, мы должны модифицировать четыре вектора. Лучшим способом было бы использовать векторы только для того, чтобы указывать направление перемещения, и использовать действительную переменную как фактор скорости — умножать перемещающий вектор на неё. Измените код, чтобы использовать переменную скорости.

Добавление камеры

У нас есть выход по клавише *Escape*, и мы можем двигать Синбада. Теперь пора вернуть обратно нашу работающую камеру.

Время для действия — делаем камеру снова работающей

Мы уже создали нашу камеру; теперь мы собираемся использовать её в комбинации с вводом пользователя следующим образом:

1. Расширьте конструктор *Framelistener*, чтобы получить указатель на нашу камеру:
```
Example30FrameListener(Ogre::SceneNode* node, RenderWindow* win,
Ogre::Camera* cam)
```
2. Также добавьте переменную-член для хранения указателя на камеру:
```
Ogre::Camera* _Cam;
```
3. Затем присвойте параметр переменной:
```
_Cam = cam;
```
4. Модифицируйте экземпляр *FrameListener*, чтобы добавить указатель на камеру:
```
Ogre::FrameListener* FrameListener = new Example30FrameListener(_SinbadNode,
mWindow, mCamera);
```
5. Чтобы перемещать камеру, нам нужно получить ввод мыши. Так что создайте новую переменную-член для хранения мыши:
```
OIS::Mouse* _mouse;
```

6. В конструкторе инициализируйте мышь после клавиатуры:
```
_mouse = static_cast<OIS::Mouse*>(_man->createInputObject( OIS::OISMouse,
false ));
```
7. Теперь, когда у нас есть мышь, нам также нужно захватывать состояние мыши, как мы это делали с клавиатурой. Добавьте эту строку после вызова захвата клавиатурного состояния:
```
_mouse->capture();
```
8. Удалите строку, перемещающую узел:
```
_node->translate(translate*evt.timeSinceLastFrame * _ movementspeed);
```
9. После обработки клавиатурного состояния в методе *frameStarted()*, добавьте следующий код для обработки состояния мыши:
```
float rotX = _mouse->getMouseState().X.rel * evt. timeSinceLastFrame* -1;
float rotY = _mouse->getMouseState().Y.rel * evt. timeSinceLastFrame * -1;
```
10. Теперь примените вращения и перемещения к камере:
```
_Cam->yaw(Ogre::Radian(rotX));
_Cam->pitch(Ogre::Radian(rotY));
_Cam->moveRelative(translate*evt.timeSinceLastFrame * _ movementspeed);
```
11. Мы создали объект мыши, так что нам нужно уничтожить его в деструкторе *FrameListener*:
```
_man->destroyInputObject(_mouse);
```
12. Скомпилируйте и запустите приложение. Вы можете двигаться по сцене так же, как и прежде.

Что произошло?

Мы использовали нашу созданную камеру в комбинации со вводом пользователя. Чтобы иметь возможность манипулировать камерой, нам нужно передать её нашему *FrameListener*. Это было сделано в шагах 1 и 2, используя конструктор. Чтобы управлять нашей камерой, мы решили использовать мышь. Так что сначала мы должны создать интерфейс мыши для использования. Это было сделан на этапе 6, так же, как мы до этого создавали клавиатуру. На этапе 7, мы были вызвали функцию *capture()* нашего нового интерфейса мыши, чтобы обновить состояние мыши.

Состояние мыши

Опрос состояния клавиатуры мы делали, используя функцию *isKeyDown()*. Чтобы получить состояние мыши, мы использовали функцию *getMouseState()*. Эта функция возвращает структуру состояния мыши как экземпляр класса MouseState, который содержит информацию о состоянии кнопок, нажаты они или нет, и о том, как мышь переместилась с тех пор, как прошел предыдущий опрос. Мы хотим по информации о перемещении

вычислить, как сильно нужно повернуть нашу камеру. Перемещение мыши может происходить по двум осям, а именно: по x и по y. Перемещения по обеим осям сохраняются отдельно в переменных X и Y состояния мыши. Кроме того, мы имеем возможность получить относительную или абсолютную величину. Поскольку мы заинтересованы только в перемещении мыши, а не её позиции, мы используем относительные величины. Абсолютные величины содержат позицию мыши на экране. Они нужны, когда мы хотим выяснить, что клик мыши произошел в определенной области нашего приложения. Для вращения камеры нам нужно только перемещение мыши, так что мы используем относительные величины. Относительная величина указывает только скорость и направление перемещения мыши, но не количество пикселей.

Затем эти величины умножаются на время, прошедшее после предыдущего кадра, и на -1. -1 используется, поскольку мы получаем перемещение в направлении, противоположном тому, куда мы хотим вращать камеру. Итак, мы просто инвертируем направление перемещения. После расчета величин вращения, мы применяем их к камере с помощью функций *yaw()* и *pitch()*. Последнее, что необходимо сделать, применить вектор перемещения, который мы создали на основе данных ввода с клавиатуры, к камере. Для этого, мы используем функцию камеры *moveRelative()*. Эта функция двигает камеру в локальном пространстве, не обращая внимания на вращение камеры. Это полезно, поскольку мы знаем, что в локальном пространстве (0,0,-1), перемещает камеру вперёд. Если же к камере были приложены вращения, то это не обязательно так. Посмотрите главу о различных пространствах в 3D для более подробного объяснения.

Популярная викторина — захват ввода

Почему мы вызываем метод *capture()* для мыши и клавиатуры?

Have a go hero — игра с примером

Попробуйте убрать -1 из вычисления поворота и посмотрите, как это изменит управление камерой.

Попробуйте удалить вызовы функции *capture()* и посмотрите, как это повлияет.

Добавление каркасного и точечного режимов рендера

В предыдущей главе, то есть, *Глава 3, Камера, Свет, и Тень*, мы использовали клавишу *R*, чтобы изменять способ рендера на каркасный или точечный. Теперь мы хотим добавить эту возможность к нашему собственному *framelistener.*

Время для действия — добавление каркасного и точечного режимов рендера

Мы используем код, который мы только что создали, и, как всегда, просто добавим код, который нам нужен для этой новой возможности:

1. Нам нужна новая переменная-член в *framelistener*, чтобы сохранять текущий режим рендера:
```
Ogre::PolygonMode _PolyMode;
```
2. Инициализируем значение в конструкторе константой *PM_SOLID*:
```
_PolyMode = PM_SOLID;
```

В оригинале книги эта строка выглядела так:
_PolyMode = Ogre::PolygonMode::PM_SOLID;

но, к сожалению, при компиляции это давало ошибку «'Ogre::PolygonMode' is not a class or namespace». После гугления выяснилось, что эту и остальные константы нужно писать просто PM_SOLID, PM_WIREFRAME, PM_POINTS. Дальше по тексту константы исправлены мной. — прим. пер.

3. Мы затем добавляем новое условное выражение *if* в функцию *frameStarted()*, которое проверяет, нажата ли клавиша *R*. Если это так, мы можем изменить режим рендера. Если текущий режим является твердотельным (solid), то мы хотим, чтобы он стал каркасным (wireframe):
```
if(_key->isKeyDown(OIS::KC_R))
{
      if(_PolyMode == PM_SOLID)
      {
            _PolyMode = PM_WIREFRAME;
      }
```
4. Если он каркасный, то мы хотим, чтобы он изменился на точечный (point) режим:
```
else if(_PolyMode == PM_WIREFRAME)
      {
            _PolyMode = PM_POINTS;
      }
```
5. И из точечного режима — обратно в твердотельный:
```
else if(_PolyMode == PM_POINTS)
      {
            _PolyMode = PM_SOLID;
      }
```
6. Теперь, когда мы вычислили новый режим рендера, мы можем применить его и закрыть выражение *if*:
```
_Cam->setPolygonMode(_PolyMode);
}
```
7. Скомпилируйте и запустите приложение; у вас должна появиться возможность изменять режим рендера нажатием клавиши *R*.

Что произошло?

Мы использовали функцию *setPolygonMode()*, чтобы изменять режим рендера из твердого тела, в каркасный, потом в точечный. Мы всегда сохраняли последнюю величину, так что при новом изменении мы знаем, какой режим является текущим, и какой должен быть следующим. Мы также должны были убедиться, что у нас все режимы зациклены. Мы меняем с твёрдотельного в каркасный, потом в точечный, и затем опять в твёрдотельный. Одна вещь, на которую надо обратить внимание — что режимы меняются довольно быстро при нажатии клавиши *R*. Дело в том, что в каждом кадре мы проверяем, нажата ли клавиша, и так как мы люди медленные, высоки шансы того, что мы нажимаем клавишу *R* дольше, чем сменяется один кадр. В результате наше приложение считает, что мы нажали клавишу *R* несколько раз в течение короткого периода времени, и переключает режимы в каждом кадре. Это не очень хорошо, и есть способ это улучшить, что мы сейчас увидим.

Добавление таймера

Решением для проблемы слишком быстрого изменения режима рендера является использованием таймера. Всякий раз, когда мы нажимаем клавишу *R*, таймер стартует, и только после того, как прошло достаточное количество времени, мы начнём обработку

другого нажатия клавиши *R*.

Время для действия — добавление таймера

1. Добавьте таймер как переменную-член к слушателю кадров (framelistener):
```
Ogre::Timer _timer;
```
2. Сбрасываем таймер в конструкторе:
```
_timer.reset();
```
3. Теперь добавьте проверку, прошли ли 0.25 секунды с тех пор, как в последний раз была нажата клавиша R. Только если это истинно, мы продолжим обрабатывать ввод:
```
if(_key->isKeyDown(OIS::KC_R) && _timer.getMilliseconds() > 250)
{
```
4. Если прошло достаточно времени, нам нужно сбросить таймер; в противном случае, клавиша *R* могла быть только что нажата:
```
_timer.reset();
```
5. Скомпилируйте и запустите приложение; теперь нажатие клавиши *R* должно изменять режим рендера только один раз.

Что произошло?

Мы использовали другой новый класс из Ogre 3D, а именно: *Ogre::Timer*. Этот класс предлагает, как и предполагается по названию, функциональность таймера. Мы сбросили (reset) таймер в конструкторе нашего слушателя, и каждый раз, когда пользователь нажимает клавишу *R*, мы проверяем, прошло ли 0.25 секунды с тех пор, как мы в последний раз вызывали *reset()*. В этом случае, мы входим в блок *if* и первая вещь, которую мы там делаем — сбрасываем таймер, а затем изменяем режим рендера, как прежде. Таким образом мы убеждается, что режим рендера изменится только после 0.25 секунды. Когда мы продолжаем нажимать клавишу *R*, мы видим как наше приложение чередует режимы рендера с ожиданием 0.25 секунд после каждого изменения.

Have a go hero — изменение режима ввода

Сделайте смену режима рендера, изменив код таким образом, чтобы режим менялся не после прошествия определенного времени, а только тогда, когда клавиша R отжималась и нажималась снова.

Итог

В этой главе мы узнали об интерфейсе *FrameListener*, и о том как его использовать. Мы также изучили, как запускать OIS, и после этого, как запрашивать состояние интерфейсов клавиатуры и мыши.

Подробнее, мы изучили:

- Как получать извещение, что рендерится новый кадр
- Важные различия между перемещениями, основанными на кадрах и на времени
- Как осуществлять наше собственное перемещение камеры, используя ввод пользователя
- Как изменять режимы рендера камеры

Теперь, когда мы осуществили основную функцию для нашего *FrameListener*, мы собираемся анимировать модели в следующей главе.

5 Анимирование моделей в Ogre 3D

Эта глава сфокусируется на анимации, на том, как она работает вообще в 3D, и в частности, в Ogre 3D. Без анимации 3D-сцена безжизненна. Это — один из наиболее важных факторов получения реалистичных и интересных сцен.

В этой главе, мы будем:

- Проигрывать анимацию
- Объединять две анимации в одно время
- Подключать сущности к анимациям

Так давайте начнём...

Добавление анимаций

В предыдущей главе мы добавили интерактивный ввод пользователя. Теперь мы собираемся добавить другую форму интерактиности к нашей сцене через анимацию. Анимации являются действительно важным фактором для каждой сцены. Без них, каждая сцена выглядит неподвижной и безжизненной, но с анимациями сцена оживает. Так давайте добавим их.

Время для действия — добавление анимации

Как всегда, мы собираемся использовать код из предыдущей главы, и на этот раз нам не нужно ничего удалять:

1. Для нашей анимации нам нужно добавить новые переменные-члены к *FrameListener*. Добавьте указатель на сущность, которую мы хотим оживить, и указатель на используемое состояние анимации:
```
Ogre::Entity* _ent;
Ogre::AnimationState* _aniState;
```
2. Затем измените конструктор *FrameListener*, чтобы получить указатель на сущность в виде нового параметра:
```
Example34FrameListener(Ogre::SceneNode* node,Ogre::Entity*
ent,RenderWindow* win,Ogre::Camera* cam)
```
3. В теле функции-конструктора назначьте данный указатель на сущность в новую переменную-член:
```
_ent = ent;
```
4. После этого извлеките состояние анимации с именем *Dance* (танец) из сущности и сохраните его в переменной-члене, которую мы создали с этой целью. Наконец, установите анимацию, которую нужно включить и зациклите её:
```
_aniState = _ent->getAnimationState("Dance");
_aniState->setEnabled(true);
_aniState->setLoop(true);
```
5. Затем нам нужно сообщить анимации, сколько времени прошло с тех пор, как она в последний раз была обновлена. Мы сделаем это в методе *frameStarted()*; в нём нам известно, сколько времени прошло:
```
_aniState->addTime(evt.timeSinceLastFrame);
```
6. Последняя вещь, которую нам нужно сделать — доработать наш *ExampleApplication* для работы с новым *FrameListener*. Добавьте новую переменную-член к приложению,

чтобы хранить указатель на сущность:
```
Ogre::Entity* _SinbadEnt;
```
7. Вместо назначения вновь созданной сущности локальному указателю, сохраните её, используя переменную-член. Замените
```
Ogre::Entity* Sinbad = mSceneMgr->createEntity("Sinbad", "Sinbad.mesh");
```
такой строкой:
```
_SinbadEnt = mSceneMgr->createEntity("Sinbad", "Sinbad.mesh");
```
8. Тоже нужно сделать при присоединении сущности к узлу:
```
_SinbadNode->attachObject(_SinbadEnt);
```
9. И конечно, при создании *FrameListener* добавьте новый параметр к вызову конструктора:
```
Ogre::FrameListener* FrameListener = new
Example34FrameListener(_SinbadNode,_SinbadEnt,mWindow,mCamera);
```
10. Скомпилируйте и запустите приложение. Вы должны увидеть танец Синбада.

Что произошло?

С помощью нескольких строк кода мы заставили Синбада танцевать. На этапе 1 мы добавили две новых переменных-члена, которые будут нужны впоследствии для анимации модели. Первая переменная-член была просто указателем на сущность, которую мы хотим оживить. Вторая была указателем на *Ogre::AnimationState*, который используется в Ogre 3D для представления одиночной анимации и связанной информации. Шаги 2 и 3 просты; мы изменили конструктор, приспособив его к новому нужному нам указателю, и на этапе 3 мы загрузили указатель в переменные-члены, которые мы создали на этапе 1. Нечто интересное происходит на этапе 4; там мы попросили сущность вернуть нам анимацию с именем *Dance* (танец). Каждая сущность сохраняет все свои анимации, и мы можем запросить их, используя строковый идентификатор и функцию *getAnimationState()*. Эта функция возвращает указатель на эту анимацию, представленное как *AnimationState*, или, если анимация отсутствует, она возвращает указатель null. После того, как мы получили состояние анимации, мы включаем его. Это укажет Ogre 3D, что нужно проиграть эту анимацию. Также, мы установили свойство loop (цикл) в истину, чтобы анимация проигрывалась снова и снова, пока мы не остановим её. Шаг 5 — важный; с этим кодом мы можем заставить анимацию ожить. Всякий раз, когда наше изображение рендерится, мы добавляем немного времени к анимации, и следовательно, Ogre 3D проиграет её немного. Если быть точным, "немного времени" соответствует времени, пройденному с предыдущего кадра. Конечно, Это можно было сделать посредством самого Ogre 3D, но такой способ более гибок. Например, мы могли бы добавить вторую модель, которую мы анимируем в замедленной съемке. Если бы Ogre 3D самостоятельно корректировал анимацию, было бы трудно или невозможно анимировать одну модель с нормальной скоростью, и одну в замедленной съемке. С используемым

методом, мы можем время, пройденное после последнего кадра, умножить на 0.25, и модель будет анимирована в замедленном темпе.

Шаги после этого — просто небольшие модификации нашего приложения, чтобы оно стало совместимым с измененным конструктором *FrameListener.* Для этого нам нужно сохранить указатель на сущность, которую мы захотели анимировать.

Популярная викторина — значение времени

Почему мы должны сообщать Ogre 3D, сколько времени прошло с момента последнего обновления анимации, и каковы положительные побочные эффекты такой архитектуры?

Have a go hero — добавление второй модели

Добавьте вторую модель, стоящую рядом с первой, и пусть она танцует в замедленном режиме.

Вы должны видеть две модели в различных стадиях одной и той же анимации, как показано на следующем рисунке.

Проигрывание двух анимаций в одно и то же время

После добавления нашей первой анимации мы собираемся посмотреть, зачем и каким образом возможно играть две анимации одновременно.

Время для действия — добавление второй анимации

Здесь мы используем тот же код, который мы использовали для первого примера:

1. Измените анимацию с *Dance* на *RunBase*.
   ```
   _aniState = _ent->getAnimationState("RunBase");
   ```
2. Скомпилируйте и Запустите приложение. Вы должны видеть бегущего Синбада, но только с нижней половиной его тела.

3. Для нашей второй анимации нам нужен новый указатель для состояния анимации:
   ```
   Ogre::AnimationState* _aniStateTop;
   ```
4. Затем, конечно, нам нужно получить анимацию для этого состояния, включить её и зациклить. Анимация, которая нам нужна, называется *RunTop*:
   ```
   _aniStateTop = _ent->getAnimationState("RunTop");
   _aniStateTop->setEnabled(true);
   _aniStateTop->setLoop(true);
   ```
5. Последнее, что нужно сделать — добавить прошедшее время к этой анимации, подобно тому, что мы делали для первой:
   ```
   _aniStateTop->addTime(evt.timeSinceLastFrame);
   ```
6. Затем снова скомпилируйте и запустите приложение. Теперь Вы должны видеть Синбада, работающего всем телом.

Что произошло?

Мы проиграли две анимации одновременно. Раньше Вы могли бы спросить, почему мы должны получать *AnimationState* для проигрывания анимации, вместо того, чтобы вызвать функцию вроде *playAnimation(AnimationName)*, теперь у вас есть на это ответ. Ogre 3D поддерживает работу нескольких анимаций за один раз, а с простым *playAnimation(AnimationName)*, так не получится. С состояниями анимации мы можем играть столько анимаций, сколько нам нужно. Мы можем даже проиграть анимации с различными скоростями, используя модифицирующую переменную и функцию *addTime()*.

Have a go hero — добавление показателя к скорости анимации

Добавьте показатель к верхней анимации и пробуйте различные величины, например 0.5 или 4.0, и посмотрите, как это повлияет на анимацию.

Давайте немного пройдемся

У нас есть анимация ходьбы, но наша модель не изменяет свою позицию. Теперь мы добавим основные элементы управления перемещением к нашей модели и смешаем их с анимациями, используя всё то, что мы уже знаем.

Время для действия — объединение пользовательского управления и анимации

И, как всегда, мы используем предыдущий код как отправной пункт:

1. Во-первых, нам нужны две новых переменных в *FrameListener* для управления скоростью перемещения и сохранения нашего вращения:
```
float _WalkingSpeed;
float _rotation;
```
2. В конструкторе мы инициализируем новые величины; мы хотим двигаться на 50 единиц в секунду и начать с состояния без вращения:
```
_WalkingSpeed = 50.0f;
_rotation = 0.0f;
```
3. Затем нам нужно изменить наши состояния анимации, чтобы предотвратить их зацикливание. На этот раз мы собираемся сами управлять запуском новой анимации, без зацикливания в Ogre 3D:
```
_aniState = _ent->getAnimationState("RunBase");
_aniState->setLoop(false);

_aniStateTop = _ent->getAnimationState("RunTop");
_aniStateTop->setLoop(false);
```
4. В методе *frameStarted()* нам нужны две новые локальные переменные, одна, чтобы указать, что мы передвинули нашу модель для этого кадра, и вторая, чтобы хранить направление, в котором мы переместили нашу модель.
```
bool walked = false;
Ogre::Vector3 SinbadTranslate(0,0,0);
```
5. Также в методе *frameStarted()* мы добавляем немного нового кода, чтобы управлять перемещением нашей модели. Мы используем клавиши позиционирования (стрелки) для перемещения. Когда клавиша нажата, нам нужно изменить переменную преобразования, чтобы сохранить направление, в котором мы хотим переместить модель, и нам нужно задать переменную вращения, чтобы повернуть модель таким образом, чтобы она смотрела в направлении перемещения:
```
if(_key->isKeyDown(OIS::KC_UP))
{
   SinbadTranslate += Ogre::Vector3(0,0,-1);
   _rotation = 3.14f;
   walked = true;
}
```
6. Нам нужно то же самое для остальных трёх клавиш позиционирования:
```
if(_key->isKeyDown(OIS::KC_DOWN))
{
      SinbadTranslate += Ogre::Vector3(0,0,1);
```

```
        _rotation = 0.0f;
        walked = true;
    }
    if(_key->isKeyDown(OIS::KC_LEFT))
    {
        SinbadTranslate += Ogre::Vector3(-1,0,0);
        _rotation = -1.57f;
        walked = true;
    }
    if(_key->isKeyDown(OIS::KC_RIGHT))
    {
        SinbadTranslate += Ogre::Vector3(1,0,0);
        _rotation = 1.57f;
        walked = true;
    }
```

7. Затем, после обработки клавиш, нам нужно проверить, двигаемся ли мы в этом кадре. Если это так, нам нужно проверить, закончилась ли анимация. В этом случае, мы перезапускаем анимацию:

```
if(walked)
{
    _aniState->setEnabled(true);
    _aniStateTop->setEnabled(true);
    if(_aniState->hasEnded())
    {
        _aniState->setTimePosition(0.0f);
    }
    if(_aniStateTop->hasEnded())
    {
        _aniStateTop->setTimePosition(0.0f);
    }
}
```

8. Если мы не двигались в этом кадре, нам нужно установить оба состояния анимации в нуль. В противном случае, наша модель окажется замороженной на полпути проведённой анимации, и это выглядело бы не очень хорошо. Так что, если мы не двигаемся в этом кадре, мы устанавливаем две анимации в стартовую позицию. Также, мы отключаем анимации, поскольку мы не перемещаем модель в этом кадре, и поэтому мы не нуждаемся в анимациях:

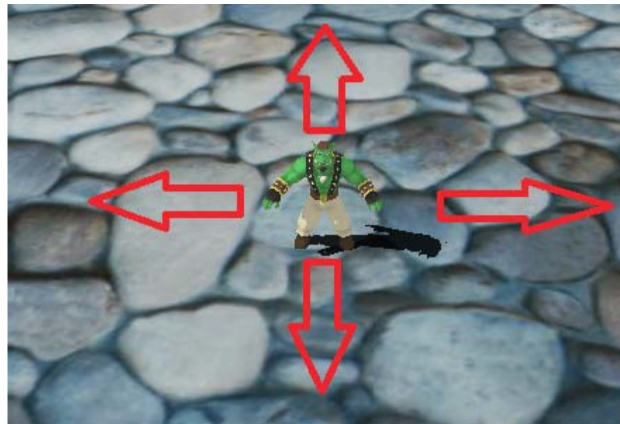
```
else
{
    _aniState->setTimePosition(0.0f);
    _aniState->setEnabled(false);
    _aniStateTop->setTimePosition(0.0f);
    _aniStateTop->setEnabled(false);
}
```

9. Последняя вещь, которую нам нужно сделать — применить перемещение и вращение к узлу сцены нашей модели:

```
_node->translate(SinbadTranslate * evt.timeSinceLastFrame *
_WalkingSpeed);
_node->resetOrientation();
_node->yaw(Ogre::Radian(_rotation));
```

10. Теперь мы компилируем и запускаем приложение. С помощью мыши и клавиш

WASD, мы можем перемещать камеру. А клавишами позиционирования мы можем перемещать Синбада, и анимация проигрывается всякий раз, когда мы перемещаем его.



Что произошло?

Мы создали наше первое приложение с вводом пользователя и комбинированием анимаций. Это можно назвать первым реальным интерактивным приложением, которое мы создали к настоящему моменту. В шагах 1 и 2 мы создали и инициализировали несколько переменных, которые нам нужны в дальнейшем. На этапе 3 мы изменили то, как мы оперируем нашими анимациями; прежде мы всегда сразу включали анимации и зацикливали их. Теперь мы не хотим, чтобы они включались автоматически, поскольку мы хотим, чтобы анимации проигрывалась только при перемещении нашей модели, иначе бы это выглядело глупо. По той же причине мы отключаем выполнение цикла анимации. Мы только хотим среагировать на ввод пользователя, так что нет необходимости для зацикливания анимации. Если нужно, мы сами запустим анимацию.

Самые большие изменения мы сделали в методе *frameStarted()*. На этапе 4 мы создали пару локальных переменных, которые нам нужны впоследствии, а именно: логическая величина, которая используется как флаг перемещения модели в этом кадре, и вектор, представляющий направление перемещения. Шаги 5 и 6 опрашивают состояние клавиш позиционирования. Когда клавиша нажата, мы изменяем вектор направления и поворот соответственно, и устанавливаем флаг перемещения в истину. Мы использовали этот флаг на этапе 7: если флаг является истиной, это означает, что наша модель перемещается в этом кадре, мы включаем анимацию и проверяем, не достигла ли анимация своего конца. Если анимация закончилась, мы перезапускаем её со стартовой позиции, чтобы она могла проиграться снова. Поскольку мы не хотим работы анимаций, когда модель не перемещается, мы устанавливаем их в стартовую позицию и отключаем на этапе 8. На этапе 9, мы применяем перемещение. Затем мы сбрасываем ориентацию, и после этого, применяем новое вращение. Это необходимо, поскольку *yaw* добавляет вращение к уже сделанным вращениям, что в нашем случае было бы неправильным, поскольку мы используем абсолютное, а не относительное вращение. Следовательно, мы сначала восстановили ориентацию, а затем применили наше вращение.

Добавление сабель

У нас теперь есть шагающая, анимированная модель, которая может управляться вводом пользователя. Теперь мы собираемся увидеть, как мы можем добавить объект к нашей анимированной модели.

Время для действия — добавление сабель

Как всегда, мы используем код из наших предыдущих упражнений:

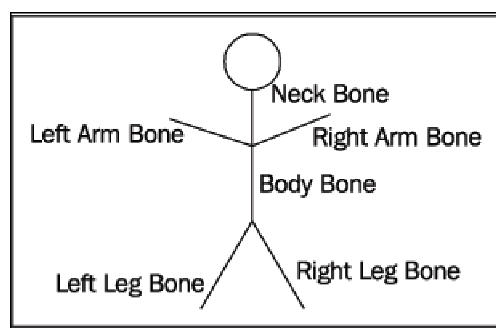
1. В конце функции *createScene()* создайте два экземпляра модели сабли и назовите их *Sword1* и *Sword2*:
```
Ogre::Entity* sword1 = mSceneMgr->createEntity("Sword1", "Sword.mesh");
Ogre::Entity* sword2 = mSceneMgr->createEntity("Sword2", "Sword.mesh");
```
2. Теперь подключите сабли к модели, используя имя костей:
```
_SinbadEnt->attachObjectToBone("Handle.L", sword1);
_SinbadEnt->attachObjectToBone("Handle.R", sword2);
```
3. Скомпилируйте и запустите приложение. В руках у Синбада должны появиться две сабли.

Что произошло?

Мы создали два экземпляра модели сабли и приложили их к костям. Создание экземпляров не должно оказаться слишком трудным для понимания. Трудной и интересной частью является функция с именем *attachObjectToBone()*. Чтобы понять, что эта функция делает, нам нужно обсудить, как анимации сохраняются и проигрываются.

Анимации

Для анимаций мы используем так называемые скелеты (skeletons) и кости (bones). Система вдохновляется от природы; в природе, почти все имеют скелет, чтобы поддерживать себя. С помощью этого скелета и мускулов животные и люди могут перемещать части своего тела в определенных направлениях. Как пример, мы можем сделать кулак из наших пальцев, используя шарниры в них. Анимации в машинной графике работают аналогичным способом. Художник определяет 3D-модель, и для этой модели создаётся скелет, так что она может быть анимирована. Скелет состоит из костей и шарниров (соединений). Шарниры соединяют две кости и определяют, в каком направлении кости могут перемещаться. Вот крайне упрощенный рисунок такого скелета; обычно, они имеют гораздо больше костей.



В модели определено, на какие треугольники влияют кости при перемещении. Каждая кость влияет только на часть модели: кость для левой руки модифицирует только треугольники модели, которые представляют левую руку; на все остальные треугольники она не воздействует.

С шарнирами, костями, и радиусом эффекта кости, художник может создавать сложные анимации, подобно анимациям для модели Синбада, которые мы используем. Также как и анимации, кости имеют имена. Ogre 3D позволяет использовать эти кости в качестве точки для подключения других сущностей. Огромное преимущество в том, что когда она приложена, сущность трасформируется подобно кости, то есть, если у нас есть точка присоединения к рукам Синбада, и приложены сабли, они всегда будут в его руках, поскольку когда руки получают трансформацию от анимации, сабли получают ту же трансформацию. Если бы эта функция не существовала, было бы почти невозможным давать модели что-то в руки или подключить что-то к ним сзади, подобно тому, как просто мы это сделали с саблями.

Печать всех анимаций, имеющихся у модели

Мы уже знаем, что художник определяет имена анимаций, но иногда важно получить имена анимаций непосредственно из Ogre 3D. Это может быть нужно, когда у вас нет художника данной модели, чтобы спросить, или когда Вы хотите проверить, что процесс экспорта был успешен.

Теперь мы увидим как вывести имена всех анимаций модели на консоль.

Время для действия — печать всех анимаций

Мы используем предыдущий код как базу, чтобы напечатать все анимации, которые имеет наша сущность.

1. В конце функции *createScene()*, получаем все анимации, которые имеет модель, в виде набора (*set*):
```
Ogre::AnimationStateSet* set = _SinbadEnt->getAllAnimationStates();
```

2. Затем определяем итератор и инициализируем его итератором набора:
```
Ogre::AnimationStateIterator iter = set->getAnimationStateIterator();
```
3. И, наконец, производим итерацию над всеми анимациями и печатаем их имена:
```
while(iter.hasMoreElements())
{
      std::cout << iter.getNext()->getAnimationName() << std::endl;
}
```
4. Скомпилируйте и запустите приложение. После запуска приложения и загрузки сцены, Вы должны увидеть следующий текст в консоли приложения:
```
Dance
DrawSwords
HandsClosed
HandsRelaxed
IdleBase
IdleTop
JumpEnd
JumpLoop
JumpStart
RunBase
RunTop
SliceHorizontal
SliceVertical
```

Что произошло?

Мы опросили сущность, чтобы получить набор, содержащий все анимации, которые она имеет. Затем мы провели итерации над этим набором и напечатали имя анимации. Мы видим, что существует множество анимаций, которые мы не использовали, а также те, что мы уже использовали.

Итог

Мы узнали много в этой главе об анимациях и о том, как их использовать, чтобы сделать 3D-сцену более интересной.

В частности, мы изучили следующее:

- Как получать анимацию из сущности и проиграть её
- Как включать/выключать и зацикливать анимации, и почему нам нужно сообщать анимации, сколько времени прошло после последнего обновления
- Как мы можем сыграть две анимации в одно и то же время
- Как анимации проигрываются, используя скелет, и что возможно подключить сущность к единственной кости
- Как запросить сущность обо всех анимациях, которые она содержит.

В следующей главе мы собираемся посмотреть на ещё один аспект Ogre 3D, главным образом на возможность использования различных менеджеров сцены и причин для такого использования.

6 Менеджеры сцены

7 Материалы

8 Композиторы пост-обработки

От переводчика:

Главы 6, 7 и 8 читайте в переводе уважаемого ***pozitiffcat****, который, как выяснилось, переводил эту книгу одновременно со мной, или даже немного раньше. Свои переводы он выкладывает на форуме по адресу http://forum.boolean.name/showthread.php?p=224502*

Прямая ссылка на перевод, работоспособная по состоянию на 01.06.2012г.: http://malcdevelop.ru/downloads/ogre_book_rus.zip

9 Последовательность запуска Ogre3D

Мы прошли большой путь в ходе этой книги. В этой главе мы охватим одну из нескольких пропущенных нами тем: как создать наши собственные приложения, не полагаясь на *ExampleApplication*. После изучения этой темы, в этой главе будут повторены некоторые темы из предыдущих глав, чтобы создать демонстрацию того, что мы узнали об использовании нашего нового открытого класса приложения.

В этой главе мы:

- Узнаем, как запускать Ogre 3D самостоятельно
- Разберём *resources.cfg*, чтобы загружать модели, которые нам нужны
- Скомбинируем всё из предшествующих глав, и создадим небольшое демонстрационное приложение, отображающее всё, что мы узнали

Давайте начнём...

Запуск Ogre 3D

До нынешнего момента класс *ExampleApplication* стартовал и инициализировал для нас Ogre 3D; теперь мы собираемся сделать это самостоятельно.

Время для действия — старт Ogre 3D

На этот раз мы работаем с чистым листом.

1. Начните с пустого файла программы, подключите *Ogre3d.h*, и создайте пустую функцию *main*:
```
#include "Ogre\Ogre.h"
int main (void)
{
      return 0;
}
```
2. Создайте экземпляр класса корня Ogre 3D Root; этот класс нуждается в имени "*plugin.cfg*":
```
Ogre::Root* root = new Ogre::Root("plugins_d.cfg");
```
3. Если диалог конфигурации не может отобразиться, или если пользователь нажал в нём отмену, закройте приложение:
```
if(!root->showConfigDialog())
{
      return -1;
}
```
4. Создайте окно рендера:
```
Ogre::RenderWindow* window = root->initialise(true,"Ogre3D Beginners Guide");
```
5. Затем создайте новый менеджер сцены:
```
Ogre::SceneManager* sceneManager = root->createSceneManager(Ogre::ST_GENERIC);
```
6. Создайте камеру и назовите её *camera*:
```
Ogre::Camera* camera = sceneManager->createCamera("Camera");
camera->setPosition(Ogre::Vector3(0,0,50));
```

```
camera->lookAt(Ogre::Vector3(0,0,0));
camera->setNearClipDistance(5);
```

7. С помощью этой камеры создайте порт просмотра и установите в нём цвет фона на черный:
```
Ogre::Viewport* viewport = window->addViewport(camera);
viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
```
8. Теперь используйте этот порт просмотра для установки соотношения ширины и высоты:
```
camera->setAspectRatio(Ogre::Real(viewport->getActualWidth())/
Ogre::Real(viewport->getActualHeight()));
```
9. Наконец, сообщите корню (экземпляру класса Root) начать рендеринг:
```
root->startRendering();
```
10. Скомпилируйте и запустите приложение; Вы должны увидеть нормальный диалог конфигурации, и затем черное окно. Это окно нельзя закрыть, нажимая *Escape*, поскольку мы пока не добавили обработку клавиш. Вы можете закрыть приложение, нажимая *CTRL+C* в консоли, с которой приложение было запущено.

Что произошло?

Мы создали наше первое Ogre 3D - приложение без помощи *ExampleApplication*. Поскольку мы больше не используем *ExampleApplication*, мы должны подключить *Ogre3D.h*, который раньше подключался через *ExampleApplication.h*. Прежде, чем мы сможем сделать что-нибудь с Ogre 3D, нам нужен экземпляр корня. Класс *root* является классом, который управляет Ogre 3D на верхнем уровне, создает и сохраняет объекты-фабрики, используемые для создания других объектов, загружает и выгружает необходимые плагины, и много чего ещё. Мы передали экземпляру корня один параметр: имя файла, в котором определено, какие плагины загружать. Следующее является полной сигнатурой его конструктора:

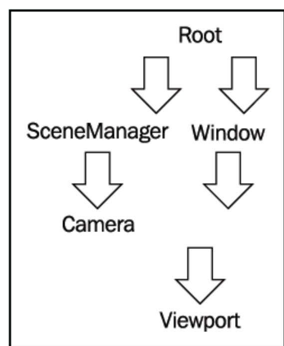
```
Root(const String & pluginFileName = "plugins.cfg", const String &
configFileName = "ogre.cfg", const String & logFileName = "Ogre.log")
```

Кроме наименования файла конфигурации плагинов, функция также нуждается в именах файла конфигурации Ogre и log-файла. Нам нужно изменить имя первого файла, поскольку мы используем отладочную версию нашего приложения, и, следовательно, хотим загрузить отладочные плагины. Значение по умолчанию - это *plugins.cfg*, который используется для каталога релизной версии Ogre 3D SDK, но наше приложение работает в отладочном каталоге, для чего используется имя файла *plugins_d.cfg*.

Файл *ogre.cfg* содержит настройки для запуска приложения Ogre, которые мы выбираем в диалоге конфигурации. Он предохраняет пользователя от необходимости изменять одно и то же в этих настройках каждый раз, который он/она запускает наше приложение. С помощью этого файла Ogre 3D может запоминать выбранные настройки и использовать их как установки по умолчанию при следующем старте. Этот файл создаётся, если он не существовал прежде, так что мы не добавляем _d к имени файла и можем использовать стандартное; то же самое истинно для log-файла.

Используя экземпляр корня, мы позволили, чтобы Ogre 3D показал диалог конфигурации пользователю в шаге 3. Когда пользователь нажимает *cancel* в диалоге, или что-нибудь идет неправильно, мы возвращаем -1 и, таким образом, закрываем приложение. В противном случае, мы создаём новое окно рендера и новый менеджер сцены на этапе 4. Используя менеджер сцены, мы создали камеру, и с ней мы создали порт просмотра; затем, используя его, мы вычислили соотношение ширины к высоте для камеры. Создание камеры и порта просмотра не должно быть для читателя чем-то новым; мы уже делали это в *Главе 3, Камера, Свет, и Тень*. После создания всего, что требовалось, мы сообщили экземпляру корня начать рендер, так что наш результат должен стать видимым. Вот диаграмма,

показывающая, какие объекты были нужны при создании другого объекта:



Добавление ресурсов

Мы создали наше первое Ogre 3D приложение, которому не нужен *ExampleApplication*. Но кое-что важное пропущено: мы пока что не загрузили и не отрендерели модель.

Время для действия — загрузка меша Синбада

У нас есть наше приложение, теперь давайте добавим модель.

1. После настройки aspect ratio и перед запуском рендера, добавьте к нашим ресурсам zip-архив, содержащий модель Синбада:
```
Ogre::ResourceGroupManager::getSingleton().addResourceLocation("../../Medi
a/packs/Sinbad.zip","Zip");
```
2. Нам больше не нужно никаких ресурсов в данный момент, так что проиндексируем все добавленные ресурсы сейчас:
```
Ogre::ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
```
3. Теперь создайте экземпляр меша Синбада и добавьте его к сцене:
```
Ogre::Entity* ent = sceneManager->createEntity("Sinbad.mesh");
sceneManager->getRootSceneNode()->attachObject(ent);
```
4. Скомпилируйте и запустите приложение; Вы должны увидеть Синбада в середине экрана:

Что произошло?

Мы использовали *ResourceGroupManager*, чтобы проиндексировать zip-архив, содержащий меш Синбада и файлы текстур, и после этого мы загрузили данные с помощью вызова *createEntity()* на этапе 3.

Использование resources.cfg

Добавление новой строки кода для каждого zip-архива или каталога, который мы хотим загрузить, является скучной задачей, и мы должны попытаться избежать этого.

ExampleApplication использовал файл конфигурации с именем *resources.cfg*, в котором был указан каждый каталог или zip-архив, и все их содержимое загружалось, используя этот файл. Давайте скопируем такое поведение.

Время для действия — использование resources.cfg для загрузки наших моделей

Используя наше предыдущее приложение, мы собираемся теперь выполнить грамматический разбор файла *resources.cfg*.

1. Замените загрузку zip-архива экземпляром config-файла, указав на *resources_d.cfg*:
```
Ogre::ConfigFile cf;
cf.load("resources_d.cfg");
```
2. Сначала получите итератор, который отсчитывает каждую секцию config-файла:
```
Ogre::ConfigFile::SectionIterator sectionIter = cf.getSectionIterator();
```
3. Определите три строки для сохранения данных, которые мы собираемся извлечь из config-файла, и повторяйте действия по каждой секции:
```
Ogre::String sectionName, typeName, dataname;
while (sectionIter.hasMoreElements())
{
```
4. Получите имя секции:
```
sectionName = sectionIter.peekNextKey();
```
5. Получите настройки, содержащиеся в секции и, в то же самое время, продвиньте итератор секции; также создайте итератор для самих настроек:
```
Ogre::ConfigFile::SettingsMultiMap *settings = sectionIter.getNext();
Ogre::ConfigFile::SettingsMultiMap::iterator i;
```
6. Повторите по каждой настройке в секции:
```
for (i = settings->begin(); i != settings->end(); ++i)
{
```
7. Используйте итератор, чтобы получить имя и тип ресурсов:
```
typeName = i->first;
dataname = i->second;
```
8. Используйте имя ресурса, тип, и имя секции, чтобы добавить его к индексу ресурсов:
```
Ogre::ResourceGroupManager::getSingleton().addResourceLocation(dataname,
typeName, sectionName);
```

Всё-таки, тут нужно закрыть скобки циклов — прим. пер.

} }

9. Скомпилируйте и запустите приложение, и Вы должны увидеть то же изображение, что и раньше.

Что произошло?

В первом шаге мы использовали другой вспомогательный класс Ogre 3D, называемый *ConfigFile*. Этот класс используется, чтобы упростить загрузку и выполнение грамматического разбора простых файлов конфигурации, которые состоят из пар имя-значение. Используя экземпляр класса *ConfigFil*е, мы загрузили *resources_d.cfg*. Мы жёстко вписали имя файла с постфиксной отладкой; это не является хорошей практикой и в промышленном приложении мы должны использовать *#ifdef*, чтобы изменять имя файла в зависимости от режима: отладки или релиза. *ExampleApplication* делает это; давайте взглянем на строку 384 файла *ExampleApplication.h*:

```
#if OGRE_DEBUG_MODE
      cf.load(mResourcePath + "resources_d.cfg");
#else
```

```
        cf.load(mResourcePath + "resources.cfg");
    #endif
```

Структура файла конфигурации

Файл конфигурации, загружаемый вспомогательным классом, следует простой структуре; вот пример из *resource.cfg*. Конечно же, ваш *resource.cfg* будет состоять из других путей:

```
[General]
FileSystem=D:/programming/ogre/ogre_trunk_1_7/Samples/Media
```

[General] начинает секцию, которая продолжается до тех пор, пока другое *[имя секции]* не встретится в файле. Каждый файл конфигурации может содержать множество секций; на этапе 2 мы создали итератор, проходящий циклом по всем секциям в файле, и на этапе 3 мы использовали цикл *while*, который выполняется, пока мы не обработали каждую секцию.

Секция состоит из нескольких настроек, и каждая настройка связывает ключ со значением. Мы назначаем ключу *FileSystem* значение *D:/programming/ogre/ogre_trunk_1_7/ Samples/Media*. На этапе 4 мы создали итератор, так что мы можем пройти циклом по каждой настройке. Внутренне настройки состоят из пар имя-значение. Мы перебираем эти отображения, и для каждого вхождения мы используем ключ, как тип ресурса, и данные, как путь. Используя имя секции как группу ресурсов, мы добавляем ресурс, используя менеджер групп ресурсов на этапе 8. Как только мы разобрали файл целиком, мы индексируем все файлы.

Создание класса приложения

У нас теперь есть основа для нашего собственного Ogre 3D приложения, но весь код находится в функции *main*, что на самом деле не желательно для повторного использования кода.

Время для действия — создание класса

Используя ранее применённый код, мы теперь собираемся создать класс, чтобы выделить код Ogre из функции *main*.

1. Создайте класс *MyApplication*, который имеет два приватных указателя, один на Менеджер сцены Ogre 3D, другой на класс *Root*:
```
class MyApplication
{
private:
      Ogre::SceneManager* _sceneManager;
      Ogre::Root* _root;
```
2. Остальная часть этого класса должна быть публичной:
```
public:
```
3. Создайте функцию *loadResources()*, которая загружает файл конфигурации *resources.cfg*:
```
void loadResources()
{
      Ogre::ConfigFile cf;
      cf.load("resources_d.cfg");
```
4. Повторяйте цикл по всем секциям файла конфигурации:
```
Ogre::ConfigFile::SectionIterator sectionIter = cf.getSectionIterator();
Ogre::String sectionName, typeName, dataname;
```

```
while (sectionIter.hasMoreElements())
{
```

5. Получите имя секции и итератор для настроек:

```
sectionName = sectionIter.peekNextKey();
Ogre::ConfigFile::SettingsMultiMap *settings = sectionIter.getNext();
Ogre::ConfigFile::SettingsMultiMap::iterator i;
```

6. Пройдите циклом по настройкам, и добавляйте каждый ресурс:

```
      for (i = settings->begin(); i != settings->end(); ++i)
      {
            typeName = i->first;
            dataname = i->second;
            Ogre::ResourceGroupManager::getSingleton().addResourceLocation(
                  dataname, typeName, sectionName);
      }
}
Ogre::ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
}
```

7. Также создайте функцию *startup()* (запуск) , которая создает экземпляр корневого класса Ogre 3D, используя *plugins.cfg*:

```
int startup()
{
      _root = new Ogre::Root("plugins_d.cfg");
```

8. Покажите диалог конфигурации, и, если пользователь выходит из него, возвратите -1, чтобы закрыть приложение:

```
if(!_root->showConfigDialog())
{
      return -1;
}
```

9. Создайте Окно рендера и Менеджер сцены:

```
Ogre::RenderWindow* window = _root->initialise(true,"Ogre3D Beginners
Guide");
_sceneManager = _root->createSceneManager(Ogre::ST_GENERIC);
```

10. Создайте камеру и порт просмотра:

```
Ogre::Camera* camera = _sceneManager->createCamera("Camera");
camera->setPosition(Ogre::Vector3(0,0,50));
camera->lookAt(Ogre::Vector3(0,0,0));
camera->setNearClipDistance(5);
Ogre::Viewport* viewport = window->addViewport(camera);
viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
camera->setAspectRatio(Ogre::Real(viewport->getActualWidth())/
Ogre::Real(viewport->getActualHeight()));
```

11. Вызовите функцию, загружающую наши ресурсы, и затем функцию, создающую сцену; после этого, Ogre 3D начинает рендерить:

```
loadResources();
createScene();
_root->startRendering();
return 0;
```

12. Затем создайте функцию *createScene()*, которая содержит код для создания узла сцены и сущности:

```
void createScene()
{
      Ogre::Entity* ent = _sceneManager->createEntity("Sinbad.mesh");
```

```
        _sceneManager->getRootSceneNode()->attachObject(ent);
}
```

13. Нам нужен конструктор, чтобы установить оба указателя на NULL, так что мы сможем удалить их, даже если им не были назначены значения:
```
MyApplication()
{
        _sceneManager = NULL;
        _root = NULL;
}
```

14. Нам нужно удалить экземпляр корня, когда наш экземпляр приложения уничтожается, так что создадим деструктор, который это сделает:
```
~MyApplication()
{
        delete _root;
}
```

15. Единственная вещь, которую осталось сделать, исправить функцию *main*:
```
int main (void)
{
        MyApplication app;
        app.startup();
        return 0;
}
```

16. Скомпилируйте и запустите приложение; изображение должно остаться неизменным.

Что произошло?

Мы переработали нашу начальный базовый код для лучшей организации по различным функциональным назначениям. Мы также добавили деструктор, так как наши созданные экземпляры должны быть удалены, когда наше приложение закрывается. Одна проблема в том, что наш деструктор не будет вызван; поскольку *startup()* никогда возвратится, нет способа закрыть наше приложение. Нам нужно добавить *FrameListener*, чтобы сообщить Ogre 3D остановить рендеринг.

Добавление FrameListener

Мы уже использовали *ExampleFrameListener*; на этот раз мы собираемся использовать нашу собственную реализацию интерфейса.

Время для действия — добавление FrameListener

Используем предыдущий код, куда мы собираемся добавить нашу собственную реализацию *FrameListener*

1. Создайте новый класс с именем *MyFrameListener*, переопределяющий три публичные функции обработки событий:
```
class MyFrameListener : public Ogre::FrameListener
{
public:
```

2. Сначала определите функцию *frameStarted*, которая сейчас возвращает ложь для закрытия приложения:
```
bool frameStarted(const Ogre::FrameEvent& evt)
{
```

```
return false;
}
```

3. Нам также нужна функция *frameEnded*, которая также возвращает ложь:
```
bool frameEnded(const Ogre::FrameEvent& evt)
{
return false;
}
```

4. Последней функцией, которую мы определим, будет функция *frameRenderingQueued*, которая также возвращает ложь:
```
bool frameRenderingQueued(const Ogre::FrameEvent& evt)
{
return false;
}
```

5. Основному классу нужно место, чтобы хранить *FrameListener*:
```
MyFrameListener* _listener;
```

6. Помните, что конструктор должен установить начальную величину *listener* в NULL:
```
_listener = NULL;
```

7. Позвольте деструктору удалить экземпляр:
```
delete _listener;
```

8. Наконец, создайте новый экземпляр *FrameListener* и добавьте его к объекту корня; это нужно сделать в функции *startup()*:
```
_listener = new MyFrameListener();
_root->addFrameListener(_listener);
```

9. Скомпилируйте и запустите приложение; оно должно сразу закрыться.

Что произошло?

Мы создали наш собственный класс *FrameListener*, который не полагается на реализацию *ExampleFrameListener*. На этот раз мы унаследовали его непосредственно от интерфейса *FrameListener*. Этот интерфейс состоит из трех виртуальных функций, которые мы реализовали. Мы уже знаем функцию *frameStarted*, но остальные две новые. Все три функции возвращают ложь, что предписывает Ogre 3D завершить рендер и закрыть приложение. Используя нашу реализацию, мы добавили *FrameListener* к экземпляру корня и запустили приложение; не было сюрприза в том, что оно тут же закрылось.

Исследование функциональности FrameListener

Наша реализация FrameListener имеет три функции; каждая вызывается в разное время. Мы собираемся исследовать, в какой последовательности они вызываются.

Время для действия — эксперимент с реализацией FrameListener

Используя печать в консоли, мы собираемся проверить, когда *FrameListener* вызывается.

1. Сначала заставим каждую функцию печатать сообщение в консоли, когда она вызывается:
```
bool frameStarted(const Ogre::FrameEvent& evt)
{
      std::cout << «Frame started» << std::endl;
```

```
        return false;
    }
    bool frameEnded(const Ogre::FrameEvent& evt)
    {
        std::cout << «Frame ended» << std::endl;
        return false;
    }
    bool frameRenderingQueued(const Ogre::FrameEvent& evt)
    {
        std::cout << «Frame queued» << std::endl;
        return false;
    }
```

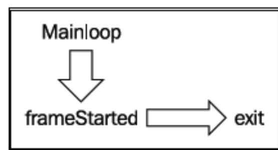
Чтобы это работало, добавьте в начало программы подключение библиотеки iostream — дополнение пер.

```
#include <iostream>
```

2. Скомпилируйте и запустите приложение; на консоли Вы должны найти первую строку — **Frame started**.

Что произошло?

Мы добавили "отладочный" вывод к каждой из функций *FrameListener*, чтобы увидеть какая функция была вызвана. При выполнении приложения мы обратили внимание, что напечатано только первое отладочное сообщение. Причина в том, что функция *frameStarted* возвращает ложь, которая является сигналом для экземпляра корня закрыть приложение.



Теперь, когда мы знаем что случается, когда *frameStarted()* возвращает ложь, давайте посмотрим что, случится, когда *frameStarted()* возвратит истину.

Время для действия — возвращаем истину в функции frameStarted()

Теперь мы собираемся модифицировать поведение нашего *FrameListener*, чтобы увидеть как оно изменится.

1. Измените *frameStarted*, чтобы она возвращала истину:

```
bool frameStarted(const Ogre::FrameEvent& evt)
{
    std::cout << «Frame started» << std::endl;
    return true;
}
```

2. Скомпилируйте и запустите приложение. Прежде, чем приложение непосредственно закроется, Вы увидите короткий проблеск отрендеренной сцены, и на вывод должно быть подано две следующих строки:

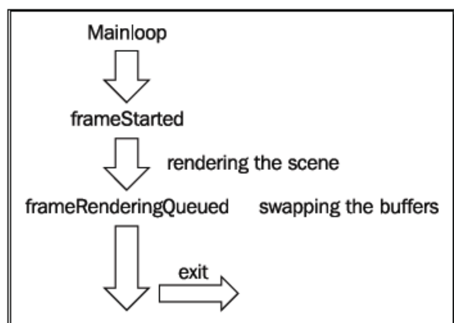
Frame started

Frame queued

Что произошло?

Теперь функция *frameStarted* возвращает истину, и это позволяет Ogre 3D продолжить

рендер, пока не будет возвращена ложь функцией *frameRenderingQueued*. На этот раз мы видим сцену, поскольку непосредственно после того, как функция *frameRenderingQueued* будет вызвана, буферы рендера поменяются перед тем, как приложение получит возможность закрыться.



Двойная буферизация

Когда сцена рендерится, она обычно не выводится непосредственно в буфер, который отображается на мониторе. Обычно, сцена рендерится во второй буфер, и когда рендер завершен, буферы меняются. Это сделано, чтобы предотвратить появление некоторых артефактов, которые могут получиться, если мы рендерим сразу в тот же буфер, который отображен на мониторе. Функция *FrameListener*'а *frameRenderingQueued* вызвается после того, как сцена отрендерится в обратный буфер, буфер, который не отображается в данное время. Прежде, чем буферы будут заменены, результат рендера уже создан, но еще не отображен. Непосредственно после того, как функция *frameRenderingQueued* будет вызвана, буферы меняются, и затем приложение получает возвращаемую величину и закрывается само. По этой причине мы видим изображение на этот раз.

Теперь мы посмотрим, что случится, когда *frameRenderingQueued* также возвратит истину.

Время для действия — возвращаем истину в функции frameRenderingQueued()

Снова мы модифицируем код, чтобы протестировать поведение Frame Listener.

1. Измените *frameRenderingQueued*, чтобы она возвращала истину:

```
bool frameRenderingQueued (const Ogre::FrameEvent& evt)
{
      std::cout << «Frame queued» << std::endl;
      return true;
}
```

2. Скомпилируйте и запустите приложение. Вы должны увидеть Синбада на короткое время, прежде, чем приложение закроется, и на выводе консоли должны появиться следующие три строки:

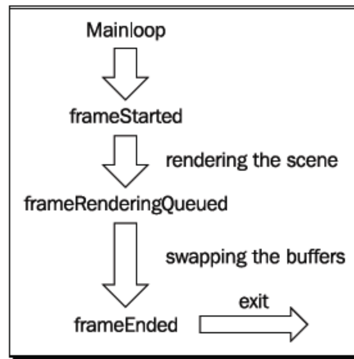
Frame started

Frame queued

Frame ended

Что произошло?

Теперь, когда обработчик *frameRenderingQueued* возвращает истину, это позволяет Ogre 3D продолжить рендер, пока обработчик *frameEnded* не вернёт ложь.



Подобно последнему примеру, буферы рендера поменялись, так что мы видели сцену в течении короткого периода времени. После того, как кадр отрендерился, функция *frameEnded* вернула ложь, что закрыло приложение и, в этом случае, с нашей точки зрения ничего не изменилось.

Время для действия — возвращаем истину в функции frameEnded()

Теперь давайте протестируем последнюю из трех возможностей.

1. Измените *frameEnded*, чтобы она возвращала истину:

```
bool frameEnded (const Ogre::FrameEvent& evt)
{
      std::cout << «Frame ended» << std::endl;
      return true;
}
```

2. Скомпилируйте и запустите приложение. Вы должны увидеть изображение Синбада и бесконечное повторение следующих трех строк:

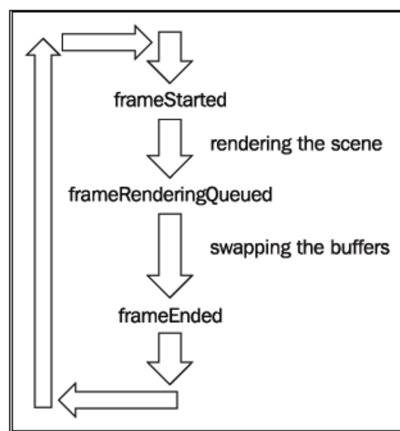
Frame started

Frame queued

Frame ended

Что произошло?

Теперь все обработчики событий возвращают истину и, следовательно, приложение никогда не будет закрыто; оно будет работать вечно, пока мы не закроем приложение сами.



Добавление ввода

У нас есть приложение, выполняющееся постоянно, и мы должны заставлять его закрыться; это не хорошо. Давайте добавим ввод и возможность закрыть приложение по нажатию *Escape*.

Время для действия — добавление ввода

Теперь, когда мы знаем, как работает *FrameListener*, давайте добавим ввод.

1. Нам нужно подлючить заголовочный файл *OIS.h*, чтобы использовать OIS:
```
#include "OIS\OIS.h"
```
2. Удалите все функции из *FrameListener* и добавьте два приватных члена, чтобы хранить InputManager и Клавиатуру:
```
OIS::InputManager* _InputManager;
OIS::Keyboard* _Keyboard;
```
3. Классу *FrameListener* нужен указатель на *RenderWindow*, чтобы инициализировать OIS, так что нам нужен конструктор, который принимает окно в качестве параметра:
```
MyFrameListener(Ogre::RenderWindow* win)
{
```
4. OIS будет инициализирован, используя список параметров, нам также нужен хендл окна в форме строки для списка параметров; создайте три необходимые переменные для хранения данных:
```
OIS::ParamList parameters;
unsigned int windowHandle = 0;
std::ostringstream windowHandleString;
```
5. Получите хендл окна рендера и преобразуйте его в строку:
```
win->getCustomAttribute("WINDOW", &windowHandle);
windowHandleString << windowHandle;
```
6. Добавьте строку, содержащую хендл окна, в список параметров, используя ключ *"WINDOW"*:
```
parameters.insert(std::make_pair("WINDOW", windowHandleString.str()));
```
7. Используйте список параметров, чтобы создать *InputManager*:
```
_InputManager = OIS::InputManager::createInputSystem(parameters);
```
8. С помощью менеджера ввода создайте клавиатуру:
```
_Keyboard = static_cast<OIS::Keyboard*>(_InputManager-
>createInputObject( OIS::OISKeyboard, false ));
```
9. То, что мы создали в конструкторе, нам нужно уничтожить в деструкторе:
```
~MyFrameListener()
{
      _InputManager->destroyInputObject(_Keyboard);
      OIS::InputManager::destroyInputSystem(_InputManager);
}
```
10. Создайте новую функцию *frameStarted*, которая захватывает текущее состояние клавиатуры, и, если нажат *Escape*, она возвращает ложь; в противном случае, она возвращает истину:
```
bool frameStarted(const Ogre::FrameEvent& evt)
{
      _Keyboard->capture();
```

```
        if(_Keyboard->isKeyDown(OIS::KC_ESCAPE))
        {
                return false;
        }
        return true;
}
```

11. Последняя вещь, которую надо сделать, это изменить реализацию экземпляра *FrameListener*, чтобы использовать указатель на окно рендера в функции запуска:
```
_listener = new MyFrameListener(window);
_root->addFrameListener(_listener);
```
12. Скомпилируйте и запустите приложение. Вы должны увидеть изображение, и теперь есть возможность закрыть окно, нажимая клавишу *Escape*.

Что произошло?

Мы добавили возможность обработки ввода в наш *FrameListener* тем же путём, что мы проделали в *Главе 4, Получение Ввода Пользователя и использование Frame Listener*. Единственное различие в том, что на этот раз мы не использовали никаких классов примеров, а только наши собственные версии.

Популярная викторина — три обработчика событий

Какие три функции предлагаются интерфейсом *FrameListener*'а, и в каком месте каждая из этих функций вызывается?

Наш собственный главный цикл

Мы использовали функцию *startRendering* для запуска нашего приложения. После этого, единственный способ управления рендером кадров, который нам доступен, возможен через *FrameListener*. Но иногда невозможно или нежелательно отказываться от управления над основным циклом; для таких случаев Ogre 3D обеспечивает другой метод, не требующий от нас отказа от контроля над основным циклом.

Время для действия — использование нашего собственного цикла рендера

Используя код, написанный ранее, мы теперь будем использовать наш собственный цикл рендера.

1. Нашему приложению надо знать, должно ли оно продолжать выполняться или нет; добавьте приватную логическую переменную-член класса приложения, чтобы помнить состояние:
```
bool _keepRunning;
```
2. Удалите вызов функции *startRendering* в функции *startup*.
3. Добавьте новую функцию, с именем *renderOneFrame*, которая вызывает функцию *renderOneFrame* экземпляра корня и сохраняет возвращаемую величину в переменной *_keepRunning*. Перед этим вызовом добавьте функцию, обрабатывающую все события окна:
```
void renderOneFrame()
{
        Ogre::WindowEventUtilities::messagePump();
```

```
        _keepRunning = _root->renderOneFrame();
}
```

4. Добавьте функцию получения значения для переменной *_keepRunning*:

```
bool keepRunning()
{
        return _keepRunning;
}
```

5. Добавьте цикл *while* в функцию *main*, который будет выполняться до момента, пока функция *keepRunning* возвращает истину. В теле цикла вызовите функцию приложения *renderOneFrame*.

```
while(app.keepRunning())
{
        app.renderOneFrame();
}
```

6. *При запуске приложения в том виде, как предлагает автор, оно сразу же закрывалось. Поразмышляв, я добавил инициализацию переменной _keepRunning в конструкторе. После этого приложение стало работать как надо. — дополнение пер.*

```
_keepRunning=true;
```

7. Скомпилируйте и запустите приложение. В нём не должно быть никаких заметных отличий от последнего примера.

Что произошло?

Мы переместили управление главным циклом из Ogre 3D в наше приложение. До этого изменения Ogre 3D использовал свой внутренний главный цикл, над которым у нас было никакого контроля, и нужно было полагаться на *FrameListener* для получения сообщений о том, что кадр отдендерен.

Теперь у нас есть наш собственный главный цикл. Чтобы попасть туда, нам нужна логическая переменная-член, которая сигнализирует, хочет ли приложение продолжать выполняться или нет; эта переменная была добавлена в шаге 1. Шаг 2 удаляет вызов функции *startRendering*, так как мы не хотим передавать управление Ogre 3D. На этапе 3, мы создали функцию, которая сначала вызывает вспомогательную функцию Ogre 3D, она обрабатывает все события окна, которые мы можем получить от операционной системы. Затем она посылает все сообщения, которые мы могли создать с момента последнего кадра, и, следовательно, делает приложение "хорошо работающим" в контексте основной оконной системы.

После этого мы вызываем функцию Ogre 3D *renderOneFrame*, которая делает в точности то, что предполагает её имя: она рендерит кадр, а также вызывает обработчики событий *frameStarted*, *frameRenderingQueued*, и *frameEnded* каждого зарегистрированного объекта *FrameListener*, и возвращает ложь, если любая из этих функций вернула ложь. Поскольку мы назначаем возвращаемое значение функции переменной *_keepRunning*, мы можем использовать эту переменную для проверки, должно ли приложение продолжать работать. Когда *renderOneFrame* возвращает ложь, мы знаем, что некоторый *FrameListener* хочет закрыть приложение, и мы устанавливаем значение переменной *_keepRunning* в ложь. Четвертый шаг просто добавляет функцию получения значения для переменной-члена *_keepRunning*.

На этапе 5, мы использовали переменную *_keepRunning* в качестве условия для цикла *while*. Это означает, что цикл будет работать так долго, пока значение переменной

_keepRunning равно истине, и что если одна из функций *FrameListener*'а вернёт ложь, это приведёт к тому, что закончится цикл *while*, и всё приложение будет закрыто. В цикле *while* мы вызываем функцию приложения *renderOneFrame*, чтобы обновить окно рендера и получить новый результат рендера. Это все, что нам нужно для создания нашего собственного главного цикла.

Добавление камеры (снова)

Мы уже реализовывали камеру в *Главе 4, Получение Ввода Пользователя и Использование Frame Listener*, но, тем не менее, мы хотим иметь управляемую камеру в нашей собственной реализации frame listener, так что здесь мы её сделаем.

Время для действия — добавление frame listener

Используя наш *FrameListener*, мы собираемся добавить камеру, управляемую пользователем.

1. Для управления камерой нам нужны следующие переменные-члены нашего *FrameListener:* интерфейс мыши, указатель на камеру, и переменная, определяющая скорость, с которой наша камера должна перемещаться:
```
OIS::Mouse* _Mouse;
Ogre::Camera* _Cam;
float _movementspeed;
```
2. Отрегулируйте конструктор: добавьте указатель на камеру, как новый параметр, и установите скорость перемещения на 50:
```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera* cam)
{
      _Cam = cam;
      _movementspeed = 50.0f;
```
3. Инициализируйте Мышь, используя *InputManager*:
```
_Mouse = static_cast<OIS::Mouse*>(_InputManager-
>createInputObject( OIS::OISMouse, false ));
```
4. И не забывайте уничтожить её в деструкторе. *Сделайте это ДО уничтожения менеджера ввода — прим. пер.*:
```
_InputManager->destroyInputObject(_Mouse);
```
5. Добавьте код перемещения камеры, использующий клавиши *W, A, S, D,* и скорость перемещения в обработчик событий *frameStarted*:
```
Ogre::Vector3 translate(0,0,0);
if(_Keyboard->isKeyDown(OIS::KC_W))
{
      translate += Ogre::Vector3(0,0,-1);
}
if(_Keyboard->isKeyDown(OIS::KC_S))
{
      translate += Ogre::Vector3(0,0,1);
}
if(_Keyboard->isKeyDown(OIS::KC_A))
{
      translate += Ogre::Vector3(-1,0,0);
}
if(_Keyboard->isKeyDown(OIS::KC_D))
```

```
{
      translate += Ogre::Vector3(1,0,0);
}
_Cam->moveRelative(translate*evt.timeSinceLastFrame * _movementspeed);
```

6. Теперь сделайте то же для управления мышью:
```
_Mouse->capture();
float rotX = _Mouse->getMouseState().X.rel * evt.timeSinceLastFrame* -1;
float rotY = _Mouse->getMouseState().Y.rel * evt.timeSinceLastFrame * -1;
_Cam->yaw(Ogre::Radian(rotX));
_Cam->pitch(Ogre::Radian(rotY));
```
7. Последняя вещь, которую надо сделать, это изменить экземпляр *FrameListener*:
```
_listener = new MyFrameListener(window,camera);
```
8. Скомпилируйте и запустите приложение. Сцена должна остаться неизменной, но теперь мы можем управлять камерой:

Что произошло?

Мы использовали наши знания из предыдущих глав для добавления управляемой пользователем камеры. Следующий шаг должен добавить композиторы и другие возможности, чтобы сделать наше приложение более интересным и использовать некоторые из методов, которые мы узнали на этом пути.

Добавление композиторов

Ранее мы создали три композитора, которые мы теперь собираемся добавить к нашему приложению, с возможностью включать и отключать каждый из них, используя ввод с клавиатуры.

Время для действия — добавление композиторов

Имея почти завершенное приложение, мы собираемся добавить композиторы, что сделает приложение более интересным.

1. Мы собираемся использовать композиторы в нашем *FrameListener*, так что нам нужна переменная-член, содержащая порт просмотра:
```
Ogre::Viewport* _viewport;
```
2. Мы также собираемся хранить информацию о том, какой композитор включен; добавьте три логические переменные для этой задачи:
```
bool _comp1, _comp2, _comp3;
```
3. Мы собираемся использовать ввод данных с клавиатуры для включения и выключения композиторов. Чтобы отличать нажатия клавиш, нам нужно знать предыдущее

состояние клавиши:

```
bool _down1, _down2, _down3;
```

4. Измените конструктор *FrameListener*'а, чтобы он получал порт просмотра в виде параметра:

```
MyFrameListener(Ogre::RenderWindow* win,Ogre::Camera* cam,Ogre::Viewport* viewport)
```

5. Присвойте указатель на порт просмотра и начальные значения логическим переменным:

```
_viewport = viewport;
_comp1 = false;
_comp2 = false;
_comp3 = false;
_down1 = false;
_down2 = false;
_down3 = false;
```

6. Если клавиша номер 1 нажата, и она не была нажата перед этим, измените состояние клавиши на нажатое, поменяйте состояние композитора, и используйте изменённую величину для включения или выключения композитора. Этот код добавляется в функцию *frameStarted*:

```
if(_Keyboard->isKeyDown(OIS::KC_1) && ! _down1)
{
      _down1 = true;
      _comp1 = !comp1;
      Ogre::CompositorManager::getSingleton().
           setCompositorEnabled(_viewport, "Compositor2", _comp1);
}
```

7. Сделайте то же самое для двух других композиторов:

```
if(_Keyboard->isKeyDown(OIS::KC_2) && ! _down2)
{
      _down2 = true;
      _comp2 = !comp2;
      Ogre::CompositorManager::getSingleton().
            setCompositorEnabled(_viewport, "Compositor3", _comp2);
}
if(_Keyboard->isKeyDown(OIS::KC_3) && ! _down3)
{
      _down3 = true;
      _comp3 = !comp3;
      Ogre::CompositorManager::getSingleton().
            setCompositorEnabled(_viewport, "Compositor7", _comp3);
}
```

8. Если клавиша больше не нажата, нам нужно изменить её состояние:

```
if(!_Keyboard->isKeyDown(OIS::KC_1))
{
      _down1 = false;
}
if(!_Keyboard->isKeyDown(OIS::KC_2))
{
      _down2 = false;
}
```

```
if(!_Keyboard->isKeyDown(OIS::KC_3))
{
	_down3 = false;
}
```

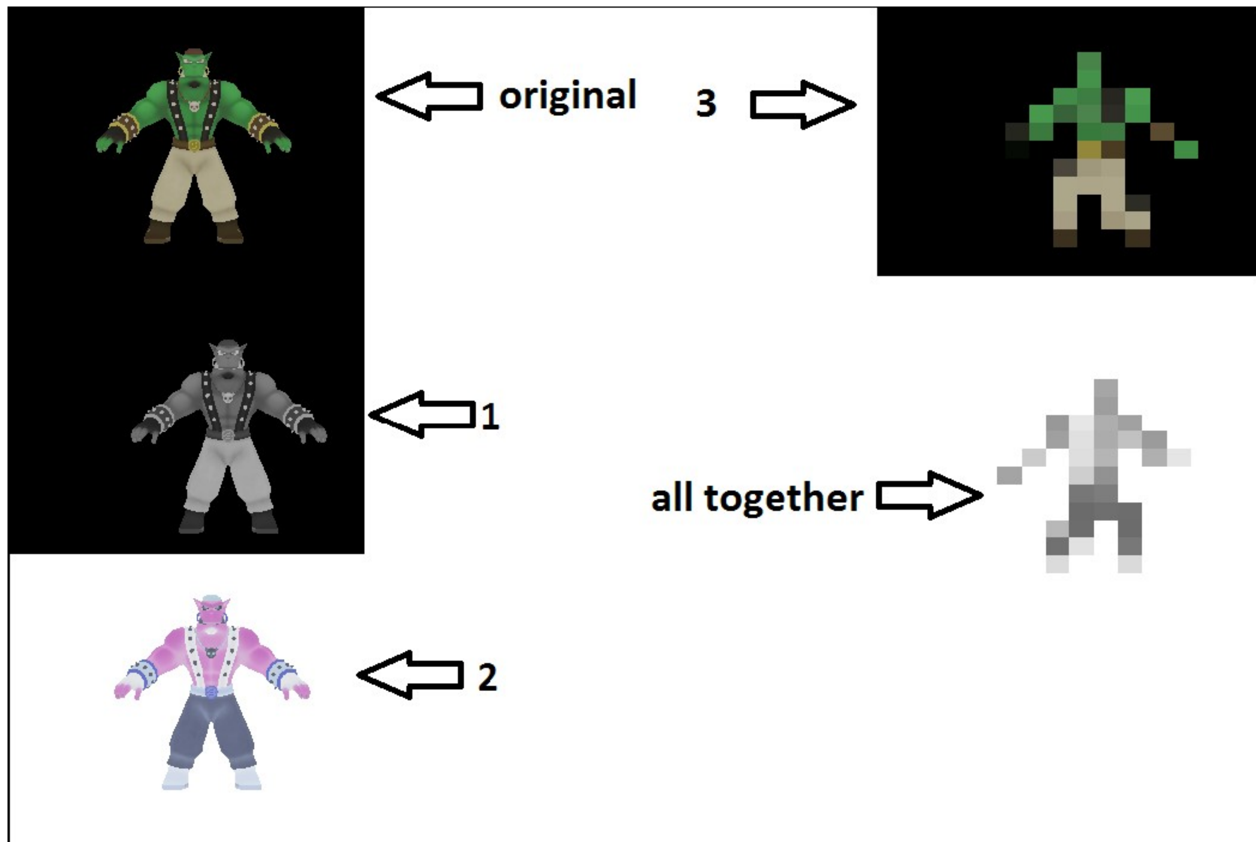
9. В функции *startup()* добавьте три композитора к порту просмотра в конце функции:

```
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor2");
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor3");
Ogre::CompositorManager::getSingleton().addCompositor(viewport,
"Compositor7");
```

10. Не забудьте изменить экземпляр *FrameListener*'а, добавив указатель на порт просмотра в виде параметра:

```
_listener = new MyFrameListener(window,camera,viewport);
```

11. Скомпилируйте и запустите приложение. Используя клавиши 1, 2, 3, Вы можете включать и выключать различные композиторы. Клавиша 1 — для получения черно-белого изображения, клавиша 2 инвертирует цвета, и клавиша 3 уменьшает разрешение у изображения; Вы можете объединять все эффекты так, как вам нравится:



Что произошло?

Мы добавили композиторы, которые мы создали в главе о них, и сделали возможным включать их, используя клавиши 1, 2, и 3. Чтобы комбинировать композиторы, мы использовали тот факт, что Ogre 3D автоматически применяет композиторы последовательно, если включено больше одного.

Добавление плоскости

Без понимания того, где находится земля, навигация в пространстве 3D затруднена, так

что давайте снова добавим плоскость пола.

Время для действия — добавление плоскости и освещения

Всё, что мы собираемся добавить на этот раз, расположено в функции *createScene()*:

1. Как мы уже знаем, нам нужно определить плоскость, так что добавим её:
```
Ogre::Plane plane(Ogre::Vector3::UNIT_Y, -5);
Ogre::MeshManager::getSingleton().createPlane("plane",
      Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
      1500,1500,200,200,true,1,5,5,Ogre::Vector3::UNIT_Z);
```
2. Затем создайте экземпляр этой плоскости, добавьте его к сцене и измените материал:
```
Ogre::Entity* ground= _sceneManager->createEntity("LightPlaneEntity",
      "plane");
_sceneManager->getRootSceneNode()->createChildSceneNode()-
      >attachObject(ground);
ground->setMaterialName("Examples/BeachStones");
```
3. Также мы хотели бы иметь какое-нибудь освещение сцены; добавьте один направленный светильник:
```
Ogre::Light* light = _sceneManager->createLight("Light1");
light->setType(Ogre::Light::LT_DIRECTIONAL);
light->setDirection(Ogre::Vector3(1,-1,0));
```
4. И тени должны выглядеть хорошо:
```
_sceneManager->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_ADDITIVE);
```
5. Скомпилируйте и запустите приложение. Вы должны видеть плоскость с каменной текстурой, и сверху неё экземпляр Синбада, отбрасывающий тень на плоскость.

Что произошло?

Снова мы использовали наши знания, приобретённые ранее, для создания плоскости, освещения, и добавления теней к сцене.

Добавление управления пользователем

У нас есть наш экземпляр модели на плоскости, но мы пока не можем перемещать его; давайте теперь это изменим.

Время для действия — управление моделью с помощью клавиш перемещения (стрелок)

Теперь мы собираемся добавить интерактивности в сцену, дав возможность пользователю управлять перемещением модели.

1. *FrameListener*'у нужно добавить два новых члена: один указатель на узел, который мы хотим перемещать, и одну переменную, содержащую скорость перемещения:
```
float _WalkingSpeed;
Ogre::SceneNode* _node;
```
2. Указатель на узел передаётся нам через конструктор:
```
MyFrameListener(Ogre::RenderWindow* win, Ogre::Camera* cam,
Ogre::Viewport* viewport, Ogre::SceneNode* node)
```
3. Присвойте указатель на узел переменной-члену и задайте скорость движения в 50:
```
_WalkingSpeed = 50.0f;
_node = node;
```
4. В функции *frameStarted* нам нужно две новых переменных, которые содержат поворот и вектор перемещения, которые пользователь хочет применить к узлу:
```
Ogre::Vector3 SinbadTranslate(0,0,0);
float _rotation = 0.0f;
```
5. Затем нам нужен код, вычисляющий перемещение и вращение в зависимости от того, какую клавишу позиционирования нажал пользователь:
```
if(_Keyboard->isKeyDown(OIS::KC_UP))
{
      SinbadTranslate += Ogre::Vector3(0,0,-1);
      _rotation = 3.14f;
}
if(_Keyboard->isKeyDown(OIS::KC_DOWN))
{
      SinbadTranslate += Ogre::Vector3(0,0,1);
      _rotation = 0.0f;
}
if(_Keyboard->isKeyDown(OIS::KC_LEFT))
{
      SinbadTranslate += Ogre::Vector3(-1,0,0);
      _rotation = -1.57f;
}
if(_Keyboard->isKeyDown(OIS::KC_RIGHT))
{
      SinbadTranslate += Ogre::Vector3(1,0,0);
      _rotation = 1.57f;
}
```
6. Затем нам нужно применить движение и вращение к узлу:
```
_node->translate(SinbadTranslate * evt.timeSinceLastFrame *
_WalkingSpeed);
_node->resetOrientation();
_node->yaw(Ogre::Radian(_rotation));
```
7. Само приложение также должно хранить указатель на узел, сущностью которого мы хотим управлять:
```
Ogre::SceneNode* _SinbadNode;
```

8. Экземпляру *FrameListener* нужен этот указатель:
```
_listener = new MyFrameListener(window,camera,viewport,_SinbadNode);
```
9. И функции *createScene* нужно использовать этот указатель для создания и хранения узла сущности, которую мы хотим перемещать; модифицируйте код функции соответственно:
```
_SinbadNode = _sceneManager->getRootSceneNode()->createChildSceneNode();
_SinbadNode->attachObject(sinbadEnt);
```
10. Скомпилируйте и запустите приложение. Вы должны иметь возможность перемещать объект с помощью клавиш со стрелками:

Что произошло?

Мы добавили перемещение сущности, используя клавиши позиционирования в *FrameListener*. Теперь наша сущность плывёт над плоскостью, словно некий волшебник.

Добавление анимации

Плавание не является именно тем, чего нам бы хотелось; давайте добавим немного анимации.

Время для действия — добавление анимации

Наша модель может перемещаться, но она пока не анимирована, давайте изменим это.

1. *FrameListener*'у нужно два состояния анимации:
```
Ogre::AnimationState* _aniState;
Ogre::AnimationState* _aniStateTop;
```
2. Чтобы получить состояния анимации в конструкторе, нам нужен указатель на сущность:
```
MyFrameListener(Ogre::RenderWindow* win, Ogre::Camera* cam,
Ogre::Viewport* viewport, Ogre::SceneNode* node, Ogre::Entity* ent)
```
3. С этим указателем мы можем извлечь нужные *AnimationState* и сохранить их для последующего использования:
```
_aniState = ent->getAnimationState("RunBase");
_aniState->setLoop(false);
_aniStateTop = ent->getAnimationState("RunTop");
_aniStateTop->setLoop(false);
```
4. Теперь, когда у нас есть *AnimationState*, нам нужен флаг в функции *frameStarted*, который сообщает нам, ходила ли сущность в этом кадре. Мы добавляем этот флаг в условия *if*, которые опрашивают состояние клавиш:
```
bool walked = false;
if(_Keyboard->isKeyDown(OIS::KC_UP))
```

```
{
      SinbadTranslate += Ogre::Vector3(0,0,-1);
      _rotation = 3.14f;
      walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_DOWN))
{
      SinbadTranslate += Ogre::Vector3(0,0,1);
      _rotation = 0.0f;
      walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_LEFT))
{
      SinbadTranslate += Ogre::Vector3(-1,0,0);
      _rotation = -1.57f;
      walked = true;
}
if(_Keyboard->isKeyDown(OIS::KC_RIGHT))
{
      SinbadTranslate += Ogre::Vector3(1,0,0);
      _rotation = 1.57f;
      walked = true;
}
```

5. Если модель перемещается, мы включаем анимацию; если анимация закончилась, мы зацикливаем её:

```
if(walked)
{
      _aniState->setEnabled(true);
      _aniStateTop->setEnabled(true);
      if(_aniState->hasEnded())
      {
            _aniState->setTimePosition(0.0f);
      }
      if(_aniStateTop->hasEnded())
      {
            _aniStateTop->setTimePosition(0.0f);
      }
}
```

6. Если модель не перемещалась, мы отключаем анимацию и устанавливаем её в стартовую позицию:

```
else
{
      _aniState->setTimePosition(0.0f);
      _aniState->setEnabled(false);
      _aniStateTop->setTimePosition(0.0f);
      _aniStateTop->setEnabled(false);
}
```

7. В каждом кадре нам нужно добавить к анимации прошедшее время; в противном случае она не будет работать:

```
_aniState->addTime(evt.timeSinceLastFrame);
```

```
_aniStateTop->addTime(evt.timeSinceLastFrame);
```

8. Приложению теперь также нужно хранить указатель на сущность:
```
Ogre::Entity* _SinbadEnt;
```
9. Мы используем этот указатель при создании экземпляра *FrameListener*:
```
_listener = new MyFrameListener(window, camera, viewport, _SinbadNode,
      _SinbadEnt);
```
10. И, конечно, при создании сущности:
```
_SinbadEnt = _sceneManager->createEntity("Sinbad.mesh");
_SinbadNode = _sceneManager->getRootSceneNode()->createChildSceneNode();
_SinbadNode->attachObject(_SinbadEnt);
```
11. Скомпилируйте и запустите приложение. Теперь модель должна шевелиться при своём перемещении:

Что произошло?

Мы добавили анимацию к нашей модели, которая включается только тогда, когда модель перемещается.

Have a go hero — ищем то, что мы использовали

Поищите главы, в которых мы рассматривали методы, используемые нами для последних примеров.

Итог

Мы много узнали в этой главе о создании нашего собственного приложения, о запуске и выполнении Ogre 3D.

В частности, мы изучили следующее:

- Как работает процесс запуска Ogre 3D
- Как сделать наш собственный главный цикл
- Написали нашу собственную реализацию приложения и *FrameListener*

Некоторые темы были уже нами изучены, но на этот раз мы объединили их для создания более сложного приложения.

Мы теперь узнали всё, что нужно для создания наших собственных Ogre 3D - приложений. Следующая глава сфокусируется на расширении Ogre 3D другими библиотеками или дополнительными возможностями, чтобы сделать приложения лучше и красивее.

10 Системы частиц и расширение Ogre3D

Это последняя глава в этой книге, в которой мы собираемся разобраться в теме, которой мы не касались до сих пор — системы частиц. После этого будут представлены некоторые возможные расширения для Ogre 3D, которые могли бы быть полезными в будущем, но не являются обязательными для каждого приложения.

В этой главе мы будем:

- Изучать, что такое система частиц, и как её можно использовать
- Создавать несколько различных систем частиц
- Знакомиться с некоторыми расширениями Ogre 3D
- Гордиться тем, что мы завершили чтение этой книги

Так давайте это сделаем...

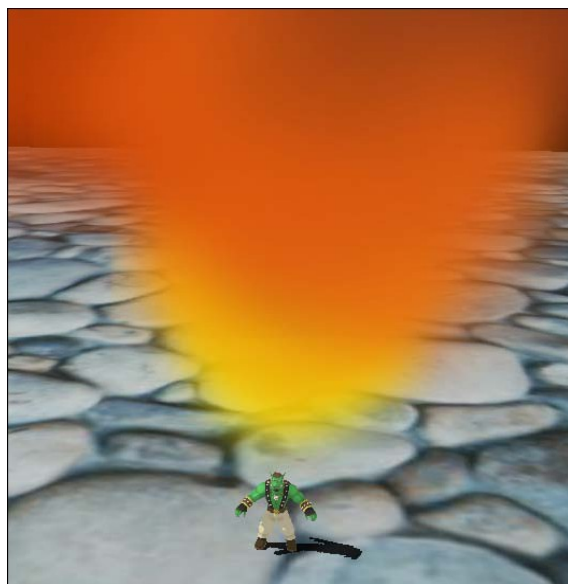
Добавление системы частиц

Мы собираемся подключить систему частиц дыма к Синбаду, так что мы всегда будем знать, где он скрыт.

Время для действия — добавление системы частиц

Мы собираемся использовать код из последнего примера:

1. Создайте систему частиц, которая использует встроенный скрипт частиц. Добавьте систему частиц к тому же узлу сцены, к которому приложена сущность Синбада:
```
Ogre::ParticleSystem* partSystem = _sceneManager->createParticleSystem( "Smoke","Examples/Smoke");
_SinbadNode->attachObject(partSystem);
```
2. Скомпилируйте и запустите приложение. Там должно быть много дыма, исходящего от Синбада.



Что произошло?

Мы использовали заранее определенный скрипт частиц для создания системы частиц, которую мы приложили к тому же узлу, к которому была приложена наша сущность. Таким образом система частиц будет следовать за нашей сущностью повсюду, при её перемещении.

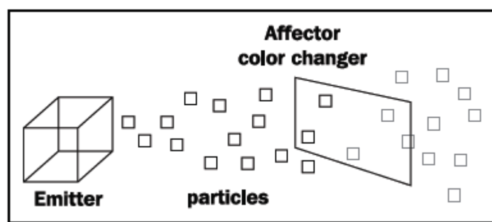
Что такое система частиц?

Прежде чем мы создадим нашу собственную систему частиц вместо загрузки встроенной, нам нужно обсудить, что в точности представляет из себя система частиц. Мы видели эффект создания системы частиц — в нашем случае, конуса дыма; но как он получился?

Система частиц состоит из двух или трех различных конструкций — эмиттера, частиц, и affector'а (опционально). Наиболее важной среди этих трёх является сама частица, как и предполагает наименование системы частиц. Частица отображает цвет или текстуру, используя способность графических карт рендерить прямоугольник или точку. Когда частица использует прямоугольник, этот прямоугольник всегда повернут лицом к камере. Каждая частица имеет набор параметров, включая время жизни, направление, и скорость. Есть множество других параметров, но эти три наиболее важные для понятия системы частиц. Параметр времени жизни регулирует жизнь и смерть частицы. Обычно частица живет не больше, чем несколько секунд, прежде чем она уничтожается. Этот эффект можно видеть в демонстрации, когда мы смотрим на конус дыма. Там есть точка, где дым исчезает. Для этих частиц счетчик времени жизни достиг нуля, и они были уничтожены.

Параметры направления и скорости описывают поведение движения частицы. В нашем случае, направлением было вверх.

Эмиттер создает заданное количество частиц в секунду и может рассматриваться как источник частиц. Affector'ы, с другой стороны, не создают частиц, но изменяют некоторые их параметры. Мы пока не видели никаких affector'ов в этой сцене, но увидим позже. Affector может изменить направление, скорость, или цвет частиц, создаваемых эмиттером.



Теперь, когда мы знаем основы, давайте создадим несколько систем частиц самостоятельно.

Создание простой системы частиц

Чтобы создать систему частиц, нам нужно определить поведение системы в целом и поведение эмиттеров в частности.

Время для действия — создание системы частиц

Мы собираемся использовать код из предшествующего примера:

1. Системы частиц определяются в файлах *.particle*. Создайте один в папке *media/particle*.
2. Определите систему и назовите её *MySmoke1*:
```
particle_system MySmoke1
```

```
{
```

3. Каждая частица должна использовать материал Example/Smoke, и её размер должен быть 10 единиц в ширину и столько же в высоту:
```
material            Examples/Smoke
particle_width      10
particle_height     10
```
4. Нам нужно максимум 500 частиц одновременно, и каждая частица должна быть точкой, которая всегда направлена на камеру:
```
quota               500
billboard_type      point
```
5. Нам нужен эмиттер, который испускает частицы из единственной точки с частотой 3 частицы в секунду:
```
emitter Point
{
      emission_rate     3
```
6. Частицы должны испускаться в направлении (1,0,0) со скоростью 20 единиц в секунду:
```
      direction         1 0 0
      velocity          20
```
7. На этом всё с этим скриптом. Закройте скобки:
```
   }
}
```
8. В функции *createScene* измените строку
```
Ogre::ParticleSystem* partSystem = _sceneManager->createParticleSystem("Smoke","Examples/Smoke");
```
на
```
Ogre::ParticleSystem* partSystem = _sceneManager->createParticleSystem("Smoke","MySmoke1");
```
9. Скомпилируйте и запустите приложение. Вы должны увидеть Синбада и дымящийся след, который возникает из него.

Что произошло?

Мы создали нашу первую собственную систему частиц. Для этого, нам нужен *.particle* файл для хранения скрипта. В этом скрипте, мы начали с определения системы частиц с помощью ключевого слова *particle_system*, и затем нам нужно присвоить ей имя, подобно тому, как мы делали для всех остальных скриптов. На этапе 3 мы определили материал, который должна использовать каждая частица. Мы использовали материал, который поставляется с SDK. Этот материал просто подключает текстуру, объединяет эту текстуру с цветом вершин, и игнорирует любое освещение. Вот полный скрипт этого материала:

```
material Examples/Smoke
{
      technique
      {
            pass
```

```
                {
                        lighting off
                        scene_blend alpha_blend
                        depth_write off
                        diffuse vertexcolor
                        texture_unit
                        {
                                texture smoke.png
                                tex_address_mode clamp
                        }
                }
        }
}
```

Мы определили длину и ширину каждой частицы в 10 единиц. Шаг 4 определяет максимальное количество частиц, которое мы хотим иметь в любой данный момент времени у системы частиц; это число полезно для предотвращения ситуации, когда неправильно определённая система частиц замедляет всё приложение целиком. Если достигнуто это число, никакому эмиттеру не будет позволено создавать новые частицы. Этот шаг также определял, что мы хотим частицы в виде точек, которые всегда направлены в камеру. Шаг 5 добавляет эмиттер, который испускает три частицы из одной точки. Шаг 6 устанавливает направление и скорость перемещения частиц. Затем мы изменили нашу программу, чтобы использовать эту новую систему частиц, и увидели её в действии.

Немного больше параметров

Теперь, когда у нас есть система частиц для экспериментов, давайте попробуем некоторые другие параметры.

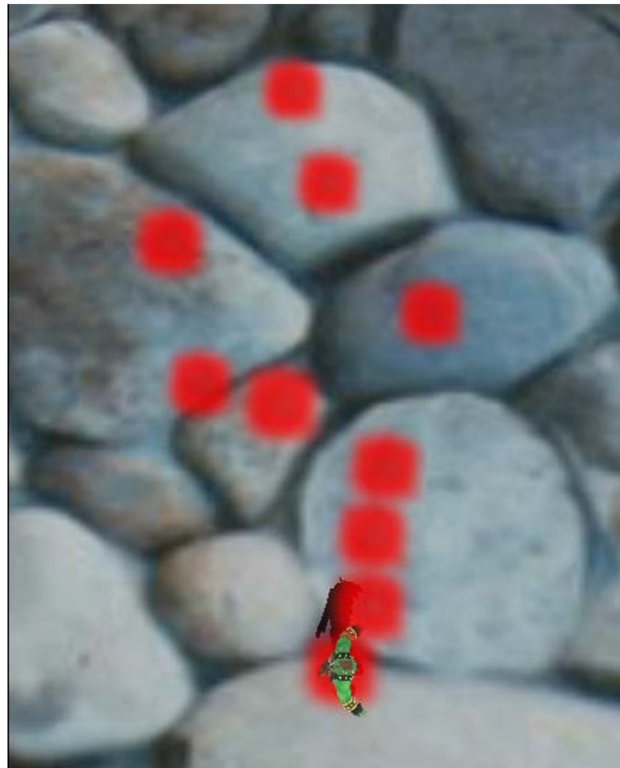
Время для действия — немного больше параметров

Мы добавим несколько новых параметров.

1. Добавьте точке эмиттера следующие три новых параметра:
   ```
   angle 30
   time_to_live 10
   colour 1 0 0 1
   ```

В оригинальном тексте здесь и далее стояло слово `color`, *но с таким параметром цвет не менялся. — прим. пер.*

2. Скомпилируйте и запустите приложение. Вы должны увидеть красные частицы, летящие в слегка различных направлениях.



Что произошло?

Мы добавили три параметра, которые изменили поведение нашей системы частиц. Теперь частицы стали красными и вылетают в различных направлениях. Параметр *angle* (угол) определяет количество градусов, на которое каждая созданная частица может отклониться от заданного направления. Ogre 3D использует генератор случайных чисел, чтобы генерировать направления из заданного диапазона. Поскольку направление может отклоняться на 30 градусов, то некоторые наши частицы могут улетать в землю.

Параметр *time_to_live* устанавливает длину жизни каждой частицы, в нашем случае 10 секунд. По-умолчанию равно 5. И, таким образом, мы удвоили жизнь каждой частицы, так что мы можем понаблюдать за их поведением дольше.

Параметр *colour* устанавливает цвет вершин частиц в соответствии с полученным вектором цвета, в нашем случае, это красный цвет.

Популярная викторина — что создаёт систему частиц

Назовите три компоненты, которые создают систему частиц; какая из их опциональна?

Другие параметры

Существует много других параметров, которые может иметь система частиц. Вот ещё несколько.

Время для действия — диапазоны для *time to live* и *colour*

Снова мы собираемся добавить несколько параметров к нашей системе частиц, чтобы увидеть эффект, который они дают.

1. Измените *time_to_live* на диапазон с минимумом и максимумом:
```
time_to_live_min 1
```

```
time_to_live_max 10
```

2. Сделайте то же самое для цвета:

```
colour_range_start 1 0 0
colour_range_end 0 0 1
```

3. Отрегулируйте код вашего приложения; затем скомпилируйте и запустите его. Вы должны видеть разноцветные частицы, и некоторые из них исчезают раньше других.

... Не очень понял последний пункт, что и зачем регулировать и компилировать? Ведь поменялся только скрипт... — прим. пер.

Что произошло?

Вместо использования параметров с единственной величиной, мы использовали параметры, которые описывали диапазон величин и позволяли Ogre 3D самому выбирать величины. Это добавляет разнообразия в нашу систему частиц и может использоваться при моделировании более достоверных естественных эффектов, поскольку в природе редко встречается что-либо, что не меняет внешний вид в пространстве и с течением времени.

Популярная викторина — time to live

Объясните своими словами различие между параметрами *time_to_live* и *time_to_live_min*.

Включение её и выключение обратно

И испытаем ещё больше параметров.

Время для действия — добавление интервалов к системе частиц

Мы теперь увидим, что есть также некоторые параметры, которые не влияют на

появление частиц, а только на способ их испускания.

1. Удалите добавленные параметры эмиттера и оставьте только *emission_rate*, *direction*, и *velocity*:
```
emitter Point
{
   emission_rate 30
   direction 1 0 0
   velocity 20
```
2. Затем добавьте параметры, которые определяют, как долго частицы должны выдаваться, и сколько времени ожидать перед началом:
```
   duration 1
   repeat_delay 1
}
```
3. Скомпилируйте и запустите приложение. Вы должны видеть поток белых частиц, который кратко прерывается всякий раз, когда эмиттер останавливается.

Что произошло?

Мы добавили параметр *duration* (длительность), который определяет сколько времени эмиттер выдаёт частицы перед остановкой. *repeat_delay* определяет время, которое ожидает эмиттер перед тем, как начать снова испускать частицы. С этими двумя параметрами, мы определили эмиттер, который одну секунду выдает частицы, затем ждет одну секунду, и начинает снова.

Популярная викторина — параметры эмиттера

Попробуйте назвать все 12 параметров эмиттера, которые мы использовали к настоящему моменту, и то, как они влияют на эмиттер.

Добавление affector'ов

Мы изменили поведение и появление частиц во время их создания, используя эмиттер. Теперь мы используем affector'ы, которые изменяют появление и поведение частицы в течении их времени жизни.

Время для действия — добавление масштабирующего affector'а

1. Чтобы показать, что делает affector, нам нужен простой точечный (*Point*) эмитттер, который выдает 30 частиц в секунду со скоростью 20 единиц и временем жизни 100 секунд:
```
emitter Point
{
   emission_rate 30
   direction 1 0 0
   velocity 20
   time_to_live 100
```

```
}
```

2. В течение всей жизни частицы мы хотим, чтобы она увеличивалась в размере в пять раз за секунду. Для этого мы добавляем масштабирующий affector *Scaler*:
```
affector Scaler
{
   rate 5
}
```
3. Скомпилируйте и запустите приложение. Вы должны видеть частицы, которые становятся больше с каждой секундой своей жизни.

Что произошло?

Мы добавили affector, который изменяет размер наших частиц в течение всей их жизни. Affector *Scaler* масштабирует каждую частицу, используя заданную величину. В нашем случае, размер каждой частицы масштабируется показателем пять каждую секунду.

Изменение цветов

Мы изменили размер. Теперь давайте изменим цвет наших частиц.

Время для действия — изменение цветов

1. Вместо масштабирования, добавьте affector *ColourFader*, который вычитает 0.25 из каждого цветового канала в секунду:
```
affector ColourFader
{
   red -0.25
   green -0.25
   blue -0.25
}
```

2. Скомпилируйте и запустите приложение. Вы должны видеть, как белые частицы становятся темнее с каждой секундой своей жизни.

Что произошло?

Мы добавили affector, который изменяет каждый цветовой канал во время существования частицы, используя заданные величины.

Have a go hero — изменение цвета на красный

Измените код у *ColourFader* так, чтобы частицы меняли цвет от белого до красного. Результат должен выглядеть похожим на это:

Двойное изменение

Мы изменили один цвет на другой, но иногда нам хотелось бы, чтобы изменение зависело от времени жизни частицы. Это может быть полезным при моделировании огня или дыма.

Время для действия — изменение в зависимости от времени жизни частицы

Теперь мы собираемся ввести больше цветов, используя affector'ы частиц.

1. Мы не хотим, чтобы наша частица жила 100 секунд в этом примере, так что изменим время жизни на 4:
```
emitter Point
{
      emission_rate 30
      direction 1 0 0
      velocity 20
      time_to_live 4
}
```
2. Поскольку мы добиваемся несколько другого поведения, мы собираемся использовать второй доступный *colorfader*. Он должен изменять каждый цветовой канал на единицу в секунду:
```
affector ColourFader2
      {
            red1 -1
            green1 -1
            blue1 -1
```
3. Теперь, когда частица просуществовала только две секунды своей жизни, вместо вычитания цветового канала, добавим ту же величину, которую мы удалили перед

этим:

```
        state_change 2
        red2 +1
        green2 +1
        blue2 +1
}
```

4. Скомпилируйте и запустите приложение.

Что произошло?

Мы использовали affector *ColorFader2*. Этот affector сначала изменяет каждую частицу исходя из величин, полученных для *red1*, *green1*, и *blue1*, когда частица живёт первые 2 секунды в соответствии с параметром *state_change*. Величины *red2*, *green2*, и *blue2* используются для модификации частиц пока они не умерли. В этом примере, мы использовали этот affector, чтобы сначала изменить частицу от белого до черного, и затем, за две секунды перед смертью, мы изменили черный на белый, следовательно создавая эффект, полученный на предыдущей картинке.

Ещё более сложные манипуляции с цветом

Существует способ создать даже более сложную обработку в отношении цвета частиц.

Время для действия — использование сложной манипуляции с цветом

Снова мы играем с цветами частиц, и с тем, как мы можем повлиять на них.

1. Мы собираемся использовать новый affector, называемый *ColorInterpolator*:

```
affector ColourInterpolator
{
```

2. Мы затем определяем, какой цвет пиксель должен иметь при его создании. Мы

используем белый:

```
time0 0
colour0 1 1 1
```

3. Когда частица прожила одну четверть своего времени жизни, она должна стать красной:

```
time1 0.25
colour1 1 0 0
```

4. В конце второй четверти жизни частицы мы хотим, чтобы она была зеленой:

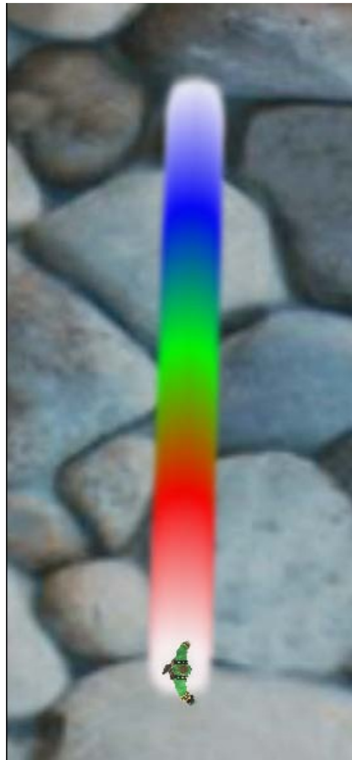
```
time2 0.5
colour2 0 1 0
```

5. В третьей четверти она должна стать синей, и, в конце, снова белой:

```
time3 0.75
colour3 0 0 1

time4 1
colour4 1 1 1
```

6. Скомпилируйте и запустите приложение, используя новый affector, — Вы должны видеть поток частиц, и их цвет должен меняться от белого на красный, на зеленый, на синий, и на белый.



Что произошло?

Мы использовали другой affector, чтобы провести более сложные манипуляции с цветом. *ColourInterpolator* манипулирует цветом всех частиц. Мы определили обработку, используя ключевые слова ***timeX*** и ***colourX***, где X должен быть между 0 и 5. *time0 0* означает, что мы хотим, чтобы affector использовал цвет *colour0* во время создания частицы. *time1 0.25* означает, что нам нужно, чтобы affector использовал *colour1*, когда частица прожила одну четверть своей жизни. Между этими двумя моментами времени affector интерполирует величины. Наш пример определяет пять точек, и каждая из них имеет отличающийся цвет. Первая и последняя точка используют белый в качестве цвета, вторая точка использует красный, третья зеленый, и четвертая синий. Точки были расположены через одну четверть

времени жизни, так что в течении жизни частицы каждый цвет был задействован одно и то же время.

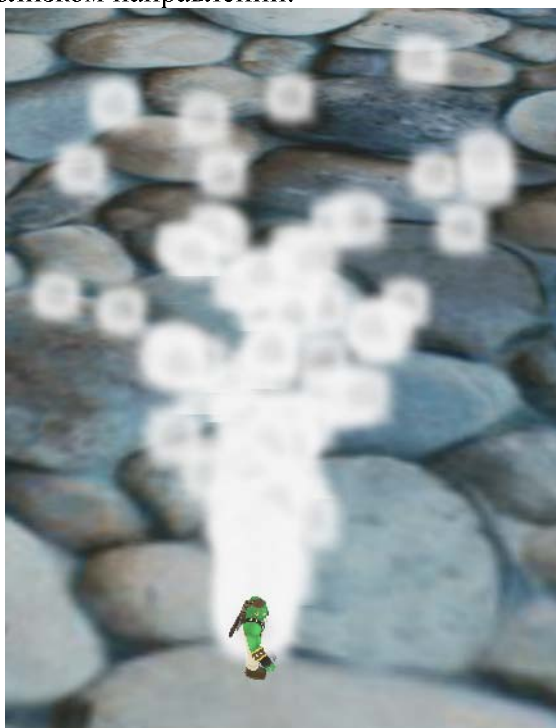
Добавление случайности

Чтобы создать красиво выглядящий эффект, иногда может помочь добавление небольшого количества произвольности в систему частиц, так, чтобы она не выглядела неестественно.

Время для действия — Добавление случайности

Добавление случайности может улучшить визуальное качество сцены, так что давайте сделаем это.

1. Удалите affector *ColourInterpolator.*
2. Добавьте другой affector с именем *DirectionRandomiser*:
```
affector DirectionRandomiser
{
```
3. Сначала мы определяем, как сильно должен влиять affector на каждую из осей наших частиц:
```
    randomness 100
```
4. Затем мы говорим, сколько наших частиц должно быть подвергнуто воздействию всякий раз, когда приложен affector. 1.0 означает 100 процентов и 0 для нуля процентов. Затем мы определяем, хотим ли мы, чтобы наши частицы сохраняли свою скорость, или её также нужно изменить:
```
    scope 1
    keep_velocity true
}
```
5. Скомпилируйте и запустите приложение. На этот раз, вы должны видеть не единственный поток, а довольно много частиц, летящих не точно по одному и тому же пути, но обычно в близком направлении.



Что произошло?

Affector *DirectionRandomiser* изменил направление наших частиц, используя различные

величины для каждой частицы. С этим affector'ом, возможно добавить компонент случайности к перемещению наших частиц.

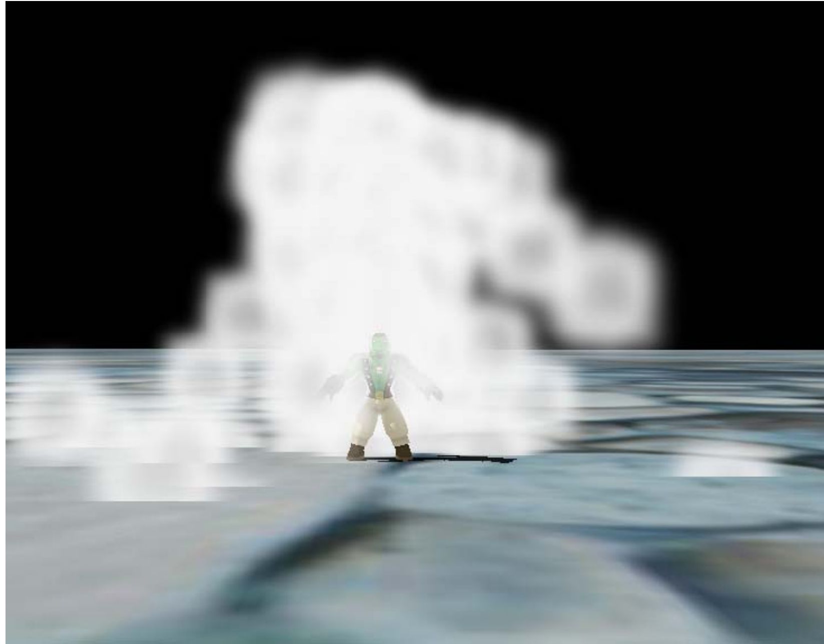
Deflector (Отклонятель)

Последний affector, который мы собираемся испытать — плоскость, которая отклоняет частицы для имитации препятствия на их пути.

Время для действия — использование отклоняющей плоскости

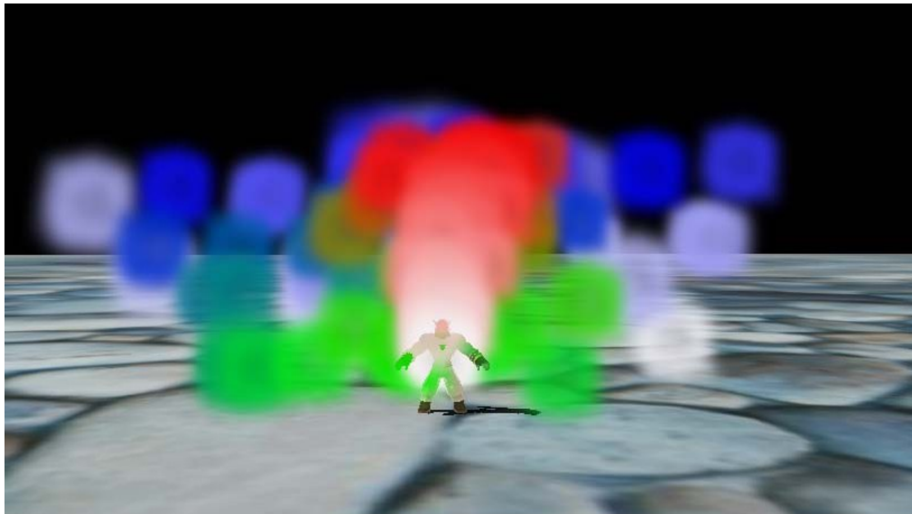
Способность заставить частицу отскакивать от некоторой поверхности может быть полезной, так что мы собираемся сделать это здесь.

1. Вместо *randomizer*, используйте affector *DeflectorPlane*:
```
affector DeflectorPlane
{
```
2. Плоскость определяется с помощью точки в пространстве и нормали к плоскости:
```
	plane_point 0 20 0
	plane_normal 0 -1 0
```
3. Последняя вещь для определения — то, как плоскость должна влиять на частицы, ударившиеся в неё. Мы хотим, чтобы они сохраняли свою первоначальную скорость, так что мы выбираем 1.0 в качестве значения:
```
	bounce 1.0
}
```
4. Чтобы увидеть эффект от плоскости *deflector*, нам нужно, чтобы наши частицы двигались в слегка различных направлениях. Так что переделаем эмиттер так, чтобы разброс направления движения частиц равнялся 30 градусам. Кроме того, так как плоскость реет в небе, у наших частиц должно быть начальное направление вверх.
```
emitter Point
{
	emission_rate 30
	direction 0 1 0
	velocity 20
	time_to_live 4
	angle 30
}
```
5. Скомпилируйте и запустите приложение. Частицы отскакивают от невидимой плоскости в небе.



Have a go hero — сделаем больше

Создайте новое приложение, где вторая плоскость в точке (0,0,0) отклоняет частицы, которые отклонились от первой плоскости. Также добавьте affector *ColourInterpolator*. Результат должен выглядеть похожим на следующий скриншот:



Другие типы эмиттеров

Мы всегда использовали точечный эмиттер для наших примеров, но, конечно же, существуют другие типы эмиттеров, которые мы можем использовать.

Время для действия — используем объёмный (box) эмиттер

Испускать только из одной точки скучно, использовать пространство гораздо веселее.

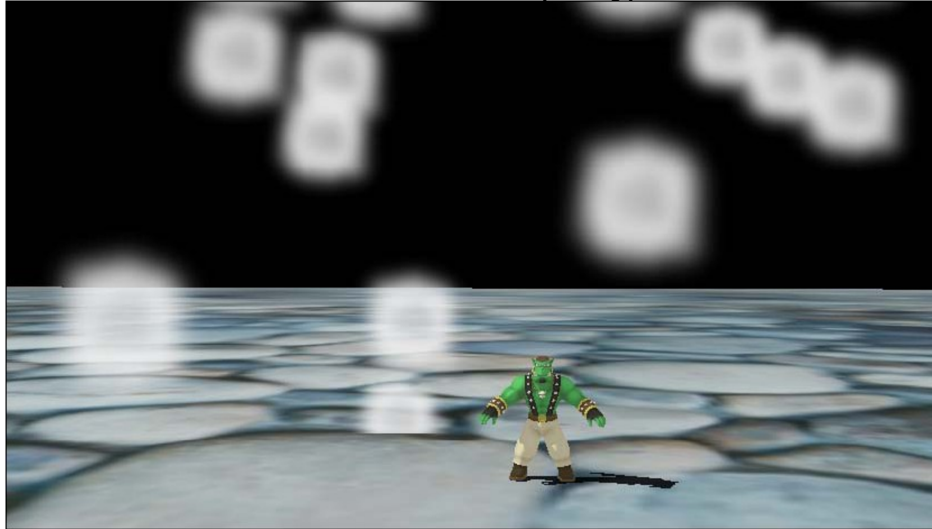
1. Измените тип эмиттера с *Point* на *Box*:
```
emitter Box
{
```
2. Определите размеры параллелепипеда, в котором должны создаваться частицы:
```
        height 50
        width 50
```

```
depth 50
```

3. Пусть эмиттер создает 10 частиц в секунду, и они должны перемещаться вверх со скоростью 20:

```
    emission_rate 10
    direction 0 1 0
    velocity 20
}
```

4. Используйте новую систему частиц. Скомпилируйте и запустите приложение. Вы должны видеть, как частицы создаются всюду вокруг Синбада и летят вверх.



Что произошло?

Мы использовали другой тип эмиттера, в данном случае эмиттер типа *Box*. Мы определили объём, и эмиттер использовал произвольные точки в этом объёме, как стартовые позиции для создания частиц. Этот эмиттер можно использовать для создания системы частиц, которая испускает не точно из одной точки, а из некоторой области. Если нам нужна просто плоскость, испускающая частицы, или даже линия, нам только нужно установить соответственным образом параметры размеров объёма.

Испускание из кольца

Кроме box'а, существуют другие типы эмиттеров, например кольцо.

Время для действия — использование кольца для испускания частиц

Вместо точки или прямоугольного объёма, мы можем даже использовать кольцо как эмиттер.

1. Измените тип эмиттера на *Ring*:

```
emitter Ring
{
```

2. Задайте параметры кольца, используя ширину и высоту:

```
    height 50
    width 50
```

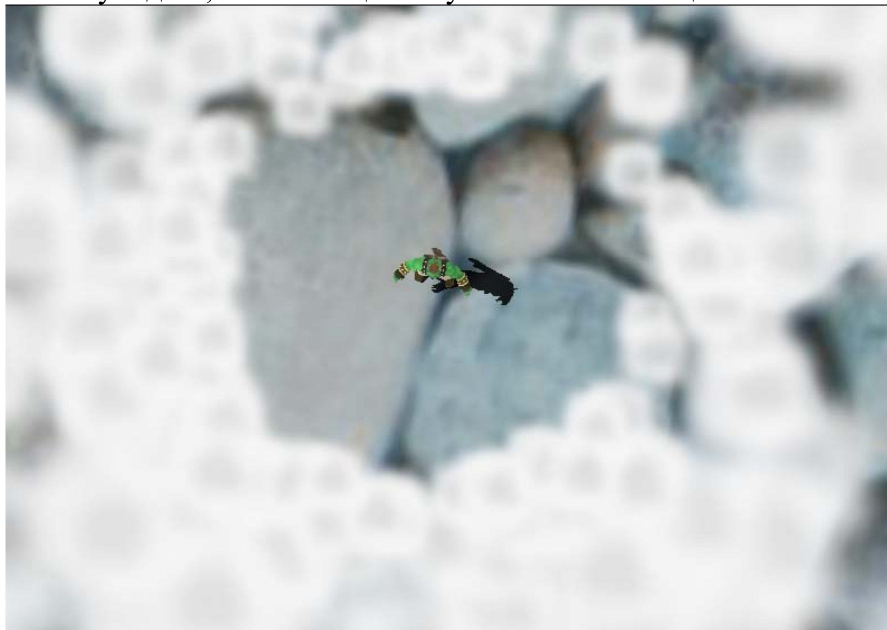
3. Теперь, чтобы создать кольцо, а не круг, нам нужно определить, какая часть внутри не должна выдавать частицы. Здесь мы используем проценты:

```
    inner_height 0.9
    inner_width 0.9
```

4. Остальное остаётся нетронутым, таким образом:

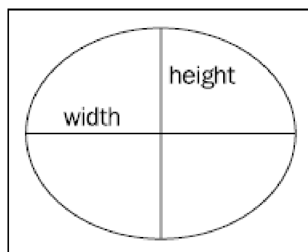
```
        emission_rate 50
        direction 0 1 0
        velocity 20
}
```

5. Скомпилируйте и запустите приложение. Подвигайте камеру над экземпляром модели, и Вы должны увидеть, что частицы испускаются из кольца.



Что произошло?

Мы использовали кольцевой эмиттер, чтобы выдавать частицы только из заданного кольца. Для того, чтобы определить кольцо, мы использовали высоту и ширину, а не точку и радиус. Width и height описывают самую большую ширину и высоту, имеющуюся у круга. Здесь, следующая небольшая диаграмма показывает, как определяется круг. (*Всё-таки правильнее его назвать эллипсом — прим. занудного переводчика*) С помощью *inner_width* и *inner_height* мы определяем, какая часть внутренней области круга не должна выдавать частицы. Здесь мы используем не абсолютные единицы размеров, а проценты.



Наконец, мы хотели бы получить фейерверк

Это будет последний пример в этой книге, так что фейерверк будет кстати.

Время для действия — добавление фейерверка

Всегда приятно видеть фейерверк после специального мероприятия.

1. Создайте систему частиц, которая взрывается разноцветными частицами во всех направлениях с постоянным интервалом:

```
particle_system Firework
{
        material Examples/Smoke
        particle_width 1
```

```
        particle_height 1
        quota 5000
        billboard_type point
        emitter Point
        {
              emission_rate 100
              direction 0 1 0
              velocity 50
              angle 360
              duration 0.1
              repeat_delay 1
              colour_range_start 0 0 0
              colour_range_end 1 1 1
        }
}
```

2. Создайте пять экземпляров этой системы частиц:

```
Ogre::ParticleSystem* partSystem1 = _sceneManager->createParticleSystem( "Firework1","Firework");
Ogre::ParticleSystem* partSystem2 = _sceneManager->createParticleSystem( "Firework2","Firework");
Ogre::ParticleSystem* partSystem3 = _sceneManager->createParticleSystem( "Firework3","Firework");
Ogre::ParticleSystem* partSystem4 = _sceneManager->createParticleSystem( "Firework4","Firework");
Ogre::ParticleSystem* partSystem5 = _sceneManager->createParticleSystem( "Firework5","Firework");
```

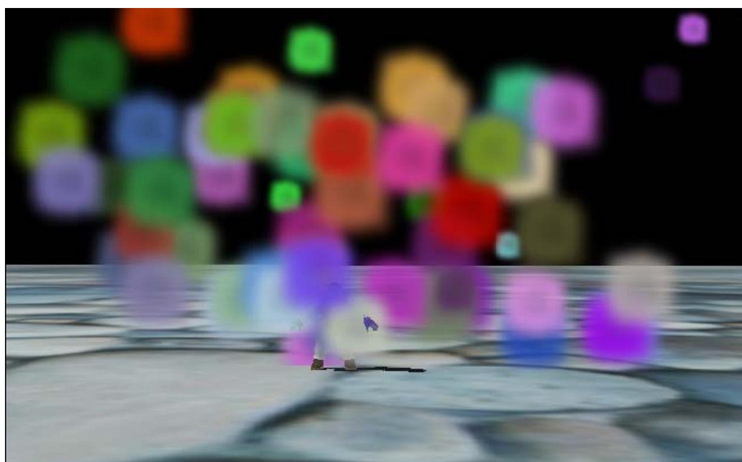
3. Затем пять узлов в различных позициях в небе:

```
Ogre::SceneNode* node1 = _sceneManager->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(0,10,0));
Ogre::SceneNode* node2 = _sceneManager->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(10,11,0));
Ogre::SceneNode* node3 = _sceneManager->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(20,9,0));
Ogre::SceneNode* node4 = _sceneManager->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(-10,11,0));
Ogre::SceneNode* node5 = _sceneManager->getRootSceneNode()->createChildSceneNode(Ogre::Vector3(-20,19,0));
```

4. Наконец, подключите системы частиц к их узлам:

```
node1->attachObject(partSystem1);
node2->attachObject(partSystem2);
node3->attachObject(partSystem3);
node4->attachObject(partSystem4);
node5->attachObject(partSystem5);
```

5. Скомпилируйте, запустите приложение в последний раз и наслаждайтесь просмотром



Что произошло?

Мы создали систему частиц, похожую на фейерверк и дублировали её так, чтобы результат выглядел похожим на несколько фейерверков в небе.

Популярная викторина — различные типы эмиттеров

Назовите все типы эмиттеров, которые мы использовали в этой главе, и несколько их различий и сходств.

Расширение Ogre 3D

Мы видели множество различных функциональных назначений, которые предлагает Ogre 3D, но Ogre 3D также позволяет его легко расширять новыми функциями. Вот почему существует большое количество различных библиотек, которые можно использовать для добавления новых функций к Ogre 3D. Мы обсудим некоторые из этих библиотек, чтобы получить представление о том, какие бывают расширения. Полный список может быть обнаружен в wiki по адресу http://www.Ogre3D.org/tikiwiki/OGRE+Libraries.

Speedtree

Speedtree — коммерческое решение, используемое, чтобы рендерить большое количество красивых деревьев и травы. Широко используется несколькими коммерческими играми, и основатель Ogre 3D Sinbad предлагает связующее ПО (binding) для Ogre 3D. Speedtree и связующее ПО для Ogre 3D должны быть куплены и не доступны свободно. Больше информации можно найти по адресу http://www.ogre3d.org/tikiwiki/OgreSpeedtree.

Hydrax

Hydrax — расширение, которое добавляет способность рендерить красивые водные сцены в Ogre 3D. С этим расширением можно добавлять воду к сцене и доступно большое количество настроек, как например, установка глубины воды, добавление эффекта пены, подводные лучи света, и так далее. Расширение можно найти по адресу http://www.ogre3d.org/tikiwiki/Hydrax.

Caelum

Caelum — другое расширение, которое вводит рендер неба с циклами дня и ночи в Ogre 3D. Оно рендерит Солнце и Луну, корректно используя дату и время. Также рендерятся эффекты погоды, подобно снегу или дождю, и сложные симуляции облаков, чтобы небо выглядело по возможности естественнее. Раздел wiki для этого расширения —

http://www.ogre3d.org/tikiwiki/Caelum.

Particle Universe

Другое коммерческое расширение — **Particle Universe** (Вселенная Частиц). Particle Universe добавляет новую систему частиц к Ogre 3D, которая даёт гораздо больше различных эффектов, чем позволяет обычная система частиц Ogre 3D. Кроме того, он поставляется с Редактором Частиц, позволяющем художникам создавать частицы в отдельном приложении, после чего программист может загружать созданный скрипт частиц позже. Это расширение можно найти в http://www.ogre3d.org/tikiwiki/Particle+Universe+plugin.

Графические интерфейсы пользователя (GUI)

Существует множество различных библиотек **GUI**, доступных для Ogre 3D, каждая из которых имеет свои причины существовать, но нет ни одной библиотеки GUI, которую должны использовать все. Лучше всего попробовать несколько из них, а затем решить для себя, какая библиотека отвечает нашим потребностям.

CEGUI

CEGUI — по-видимому первая библиотека графического интерфейса пользователя, которая была интегрирована в Ogre 3D. Она предлагает все функции, ожидаемые от системы GUI, и много больше. Есть GUI-редактор для создания вашего интерфейса вне кода и множество различных скинов для модификации по заказу пользователя вашего интерфейса. Больше информации можно найти по адресу http://www.cegui.org.uk/wiki/index.php/Main_Page.

BetaGUI

BetaGUI — очень маленькая библиотека, которая состоит из одного заголовочного и одного cpp-файла. Единственная её зависимость — Ogre 3D, и она предлагает базовую функциональность, подобно созданию окна, кнопок, текстовых областей, и статического текста. Она не является полноценным графическим интерфейсом пользователя, но предлагает базовую функциональность без зависимостей, так что может быть использована, когда нужно простое и быстрое решение. Больше можно прочитать по адресу http://www.ogre3d.org/tikiwiki/BetaGUI.

QuickGUI

QuickGUI — более сложное и мощное решение, чем BetaGUI. Хотя QuickGui предлагает гораздо больше виджетов, её процесс интеграции также немного сложнее. QuickGUI — полноценное GUI-решение, которое может быть использовано для самых разных проектов и регулярно обновляется. Вики-сайт можно найти по адресу http://www.ogre3d.org/tikiwiki/QuickGUI.

Berkelium

Berkelium не является GUI-библиотекой, как таковой, так как у неё нет никаких виджетов или чего-нибудь похожего. Вместо этого она позволяет Ogre 3D рендерить веб-сайты, используя библиотеку Google Chromium. С помощью этой библиотеки возможно построить внутриигровой веб-браузер. Веб-сайт можно найти по адресу в http://www.ogre3d.org/tikiwiki/Berkelium.

Итог

Мы узнали много в этой главе.

В частности, мы изучили:

- Как создавать систему частиц, используя различные типы эмиттеров
- Как affector'ы могут влиять на частицы
- Какие расширения доступны для Ogre 3D

Конец

Это конец книги, и я хотел бы Вас поздравить. Я знаю, что это — большая работа, читать целиком книги по программированию и выполнять все примеры для понимания новой темы, но она также на самом деле вознаграждается, и новое знание навсегда останется вашим. Я надеюсь, что Вы насладились этой книгой, и изучили достаточно, чтобы самостоятельно создавать ваши собственные интерактивные 3D-приложения, поскольку, по-моему, это одна из наиболее интересных и быстро изменяющихся областей программирования и информатики в целом.