А. В. Казанцев

Основы компьютерной графики для программистов

Часть 1. Математические основы компьютерной графики

Часть 2. Приложения компьютерной графики (Win32 и OpenGL)

v.1.01 - Казань 2005

Содержание

Учебное пособие создано на основе специальных курсов лекций, читаемых автором с 1996 года в Казанском Государственном Университете на факультете вычислительной математики и кибернетики, и является развитием первой версии текстов лекций “Основы компьютерной графики” 2001 года. Предназначено для начинающих осваивать компьютерную графику. Здесь содержатся базовые знания необходимые для разработки двумерных и трехмерных приложений компьютерной графики. Исправлены некоторые ошибки и недочеты, найденные в Части 1 предыдущей версии, а также дополнены ее некоторые главы. В Части 2 содержится введение в оконный интерфейс Win32 и отдельные главы графической библиотеки OpenGL. Эти лекции находятся в сети Интернет по адресу, указанному в нижнем колонтитуле. Там же можно найти архив проекта программы на Delphi, о которой идет речь в главах, посвященных моделям освещенности в OpenGL. Ваши сообщения о замеченных в тексте опечатках, и другие замечания можно присылать по адресу электронной почты Alex.Kazantsev@ksu.ru .

Введение

Во многих книгах по компьютерной графике исследуются узкоспециальные темы, такие как разработка библиотек подпрограмм для реализации метода обратного хода лучей, низкоуровневое программирование видеоадаптеров или описание скоростных методов изображения трехмерных сцен, которые используются в компьютерных играх. При этом, для тех, кто только начинает вникать в эту область, часто не хватает информации базового уровня, позволяющей сориентироваться в стремительно расширяющейся области компьютерной графики. Данный материал призван, хотя бы отчасти, восполнить указанный пробел.

Отправной точкой зарождения компьютерной графики можно считать 1930 год, когда в США нашим соотечественником Владимиром Зворыкиным, работавшим в компании “Вестингхаус” (Westinghouse), была изобретена электронно-лучевая трубка (ЭЛТ), впервые позволяющая получать изображения на экране без использования механических движущихся частей. Именно ЭЛТ является прообразом современных телевизионных кинескопов и компьютерных мониторов. Началом эры собственно компьютерной графики можно считать декабрь 1951 года, когда в Массачусетском технологическом институте (МТИ) для системы противовоздушной обороны военно-морского флота США был разработан первый дисплей для компьютера “Вихрь” (Whirl). Изобретателем этого дисплея был Джей Форрестер, работавший инженером в МТИ.

Одним из отцов-основателей компьютерной графики считается Айвен Сазерленд (Ivan Sotherland), который впервые в 1962 году все в том же МТИ создал программу компьютерной графики под названием “Блокнот” (Sketchpad). Эта программа могла рисовать достаточно простые фигуры (точки, прямые, дуги окружностей), могла вращать фигуры на экране. После этой программы некоторые крупные фирмы, такие как “Дженерал моторз”, “Дженерал электрик”, приступили к разработкам в области компьютерной графики. В 1965 году фирма IBM выпустила первый коммерческий графический терминал под названием IBM-2250. В конце 70-х годов для космических кораблей “Шаттл” появились летные тренажеры, основанные на компьютерной графике. Такие тренажеры представляют собой полнофункциональную модель кабины космического корабля, у которой вместо окон установлены компьютерные мониторы. На этих мониторах синтезируется изображение, которое видят астронавты из взлетающего космического корабля. В 1979 году Джордж Лукас, создатель сериала

“Звездные войны”, организовал в своей фирме “Lucasfilm” отдел, который занимался внедрением последних достижений компьютерной графики в кинопроизводство. В 1982 году на экраны кинотеатров вышел фильм “Трон”, в котором впервые использовались кадры, синтезированные на компьютере.

Существуют фирмы, специализирующиеся на разработке специализированных компьютеров для графических приложений, такие как “Silicon Graphics”, “Evans&Sotherland”. Области приложения компьютерной графики в настоящее время очень широки. В промышленности используется компьютерное моделирование процессов с графическим отображением происходящего на экране. Разработка новых автомобилей проходит на компьютере от стадии первичных эскизов внешнего вида корпуса автомобиля до рассмотрения поведения деталей автомобиля в различных дорожных условиях. В медицине применяются компьютерные томографы, позволяющие заглянуть внутрь тела и поставить правильный диагноз. В архитектуре широко применяются системы визуального автоматизированного проектирования (CAD – Computer Aided Design) которые позволяют разработать проект нового здания, основываясь на методах компьютерной графики. Химики изучают сложные молекулы белков, пользуясь компьютерными средствами визуального отображения данных. В телевидении и кинематографии использование компьютерной графики стало почти необходимым делом. В мире регулярно проводятся выставки, например, такие как SIGGRAPH, картин нарисованных с помощью компьютера. В математике развитие теории фракталов было бы невозможно без компьютеров с соответствующими средствами графического отображения данных. Средства мультимедиа привели к возможности совместного использования различных источников информации, объединяющих в себе статические и видео изображения, текст и звук. Новейшие операционные системы работают в графическом режиме и изначально реализуют в своих функциях методы компьютерной графики.

ЧАСТЬ 1. МАТЕМАТИЧЕСКИЕ ОСНОВЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

Глава 1. Элементы аналитической геометрии

Специфика математического аппарата компьютерной графики состоит в его исключительно практической направленности. Математические методы компьютерной графики предназначены для получения зрительно осязаемых результатов. Однако использование прикладных математических методов не освобождает от знания теоретических основ, из которых эти методы были получены. В данной главе рассматриваются элементы теории аналитической геометрии в трехмерном пространстве как поэтапное построение теоретических конструкций, происходящих из необходимости решения некоторых практических задач. Такой, в некотором смысле, неформальный подход позволяет рассматривать аналитическую геометрию не просто как раздел линейной алгебры, а как мощную методологию решения практических геометрических задач, возникающих в трехмерных и двумерных приложениях компьютерной графики.

Система координат

Для того чтобы уметь синтезировать изображения на экране компьютера необходимо предложить способ математического описания объектов в трехмерном пространстве или на плоскости. Окружающий нас мир с точки зрения практических приложений описывают как трехмерное евклидово пространство. Под описанием трехмерного объекта будем понимать знание о положении каждой точки объекта в пространстве в любой момент времени. Положение точек в пространстве удобно описывается с помощью декартовой системы координат.

Для того чтобы в трехмерном пространстве задать декартову систему координат проведем три не лежащие в одной плоскости направленные прямые, которые

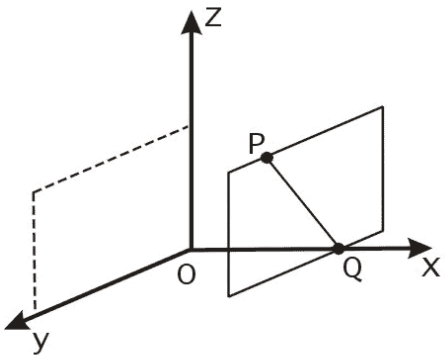

Рис. 1. Нахождение координаты $x = Q$ точки P.

называются осями, так, чтобы они пересекались в одной точке – начале координат. Выберем на этих осях единицу измерения. Тогда положение любой точки в пространстве будет описываться через координаты этой точки, которые представляют

собой расстояния от начала координат до проекций точки на соответствующие оси координат. Проекцией точки на координатную ось называется точка пересечения плоскости, проходящей через заданную точку и параллельной плоскости, образованной двумя другими осями координат. Например, на рис. 1 проекцией точки P на ось Ox

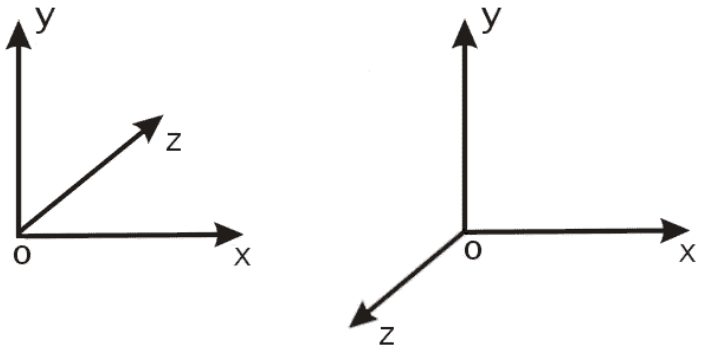

Рис. 2. Левосторонняя и правосторонняя системы координат.

является точка Q, которая принадлежит плоскости, параллельной плоскости yOz.

В общем случае оси системы координат могут располагаться под произвольными, хотя и фиксированными углами друг относительно друга. Для практических расчетов гораздо удобнее, когда эти оси расположены взаимно перпендикулярно. Такая система координат называется ортогональной. В ортогональной системе координат проекцией точки P на ось является единственная точка на оси такая, что отрезок прямой, проведенной из этой точки к точке P, является перпендикулярным к данной оси.

Таким образом, положение в пространстве точки P описывается ее координатами, что записывается как $P = (p_x, p_y, p_z)$. Взаимное расположение осей в ортогональной системе координат в трехмерном пространстве может быть двух видов. Проведем ось Ox слева направо, а ось Oy снизу вверх, как показано на рис. 2.

Ось Oz при этом может проходить как в направлении от наблюдателя в плоскость листа, так и от плоскости листа к наблюдателю. В первом случае система координат будет называться левой или левосторонней, а во втором случае – правой или правосторонней. Более точное определение правой и левой систем координат можно дать следующее. Если посмотреть из положительной полуоси Oz в направлении начала координат, то для совмещения положительной полуоси Ox с положительной полуосью Oy необходимо повернуть Ox относительно начала координат против часовой стрелки – в этом случае имеем правую систему координат; если же поворот

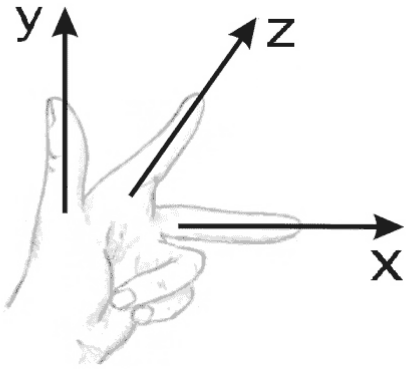

Рис. 3. Определение левосторонней системы координат по левой руке.

производится по часовой стрелке – то система координат левая*.

Существует также легкий способ определения вида системы координат по правой или левой руке, как показано на рис. 3. Для левой руки большой, указательный и средний пальцы формируют левую тройку ортогональных векторов. То же относится и к их циклическим перестановкам.

Декартовы координаты точек позволяют описывать статичное положение точек в пространстве. Однако, для проведения каких-либо преобразований над объектами, которые описываются точками, необходимо иметь дополнительный математический аппарат. В качестве такого математического аппарата применяют радиус-векторы. Радиус-векторы обладают всеми свойствами векторов, но имеют одну особенность: начало радиус-вектора находится всегда в начале системы координат, а конец радиус-вектора лежит в некоторой точке пространства. Это свойство радиус-векторов позволяет поставить во взаимно однозначное соответствие всем точкам пространства соответствующие им радиус-векторы. Формально это соответствие запишем в следующем виде. Пусть точка P имеет координаты (p_x, p_y, p_z), то есть $P = (p_x, p_y, p_z)$, и $\boldsymbol{p} = p_x\boldsymbol{i} + p_y\boldsymbol{j} + p_z\boldsymbol{k}$ – радиус-вектор, конец которого находится в точке P, где $\boldsymbol{i,j,k}$ – тройка единичных базисных векторов (ортов), или просто ортонормированный базис. Тогда точке P взаимно однозначно соответствует радиус-вектор p, или $P = (p_x, p_y, p_z) \Leftrightarrow p_x\boldsymbol{i} + p_y\boldsymbol{j} + p_z\boldsymbol{k} = \boldsymbol{p}$. Таким образом, можно легко переходить от координат точек к радиус-векторам и обратно. Далее мы увидим, что представление радиус-вектора в виде линейной комбинации векторов базиса имеет вполне конкретное практическое применение. Отметим, что радиус-вектор иногда определяют как преобразование переноса точки из начала координат в заданную точку пространства с известными координатами. При этом умножение радиус-вектора p на число a соответствует переносу точки из начала координат в направлении вектора p на расстояние $a|\boldsymbol{p}|$, где в прямых скобках $|\boldsymbol{p}| = \sqrt{{p_x}^2 + {p_y}^2 + {p_z}^2}$ – модуль вектора.

Сложение радиус-векторов $\boldsymbol{p}+\boldsymbol{q}$ можно рассматривать как перенос точки P по направлению вектора $\boldsymbol{q}$ на расстояние $|\boldsymbol{q}|$.

Уравнение прямой

Рассмотрим, каким образом можно использовать координаты точек и радиус-векторы для описания прямых в трехмерном пространстве. Уравнение прямой дает информацию, принадлежит ли точка с заданными координатами определенной прямой или нет. Рассмотрим два способа вывода этого уравнения. В первом случае выберем в пространстве две точки $P_1 = (x_1, y_1, z_1) \Leftrightarrow \boldsymbol{p}_1$ и $P_2 = (x_2, y_2, z_2) \Leftrightarrow \boldsymbol{p}_2$.

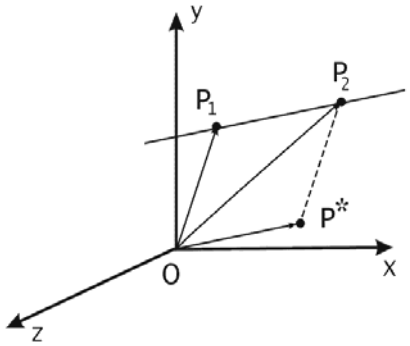

Рис. 4. Вывод уравнения прямой в трехмерном пространстве.

Проведем от точки P_1 к точке P_2 обычный вектор

* В этом определении при замене, скажем, оси Oz на ось Ox остальные оси заменяются по правилу циклической перестановки, то есть Oy заменится на Oz, а Ox заменится на Oy. Всего здесь циклических перестановок может быть три: (x,y,z)→(y,z,x)→(z,x,y).

равный разности векторов $\boldsymbol{p}_2 - \boldsymbol{p}_1$. Этому вектору соответствует параллельный ему радиус-вектор $\boldsymbol{p}^* = \boldsymbol{p}_2 - \boldsymbol{p}_1$, как показано на рис. 4. Тогда радиус-вектор $\boldsymbol{p}$, определяющий некоторую точку на прямой, можно получить сложением, например, вектора $\boldsymbol{p}_1$ и вектора $\boldsymbol{p}^*$, умноженного на некоторое число μ:

$$\boldsymbol{p} = \boldsymbol{p}(\mu) = \boldsymbol{p}_1 + \mu\boldsymbol{p}^* .$$

Так мы получили уравнение прямой в векторной форме, с помощью, так называемых, базового и направляющего векторов, $\boldsymbol{p}_1$ и $\boldsymbol{p}^*$, соответственно. Преобразуем это уравнение к виду, в котором используются только координаты двух исходных векторов.

$$\boldsymbol{p} = \boldsymbol{p}_1 + \mu\boldsymbol{p}^* = \boldsymbol{p}_1 + \mu(\boldsymbol{p}_2 - \boldsymbol{p}_1) \Rightarrow \boldsymbol{p} - \boldsymbol{p}_1 = \mu(\boldsymbol{p}_2 - \boldsymbol{p}_1) \qquad (1)$$

Из этого векторного равенства получаем три равенства для соответствующих координат:

$$\begin{cases} x - x_1 = \mu(x_2 - x_1) \\ y - y_1 = \mu(y_2 - y_1) \\ z - z_1 = \mu(z_2 - z_1) \end{cases}$$

Попарно разделив эти уравнения друг на друга, получаем следующую систему уравнений, определяющую нашу прямую в трехмерном пространстве в форме записи через координаты точек:

$$\begin{cases} (x - x_1)(y_2 - y_1) = (x_2 - x_1)(y - y_1) \\ (y - y_1)(z_2 - z_1) = (y_2 - y_1)(z - z_1) \\ (z - z_1)(x_2 - x_1) = (z_2 - z_1)(x - x_1) \end{cases} \qquad (2)$$

В практических задачах иногда бывает нужно узнать, лежит ли некоторая точка, принадлежащая прямой, внутри отрезка, заданного координатами своих концов, на данной прямой или снаружи. Для решения этой задачи перепишем уравнение (1) в следующем виде:

$$\boldsymbol{p} = (1 - \mu)\boldsymbol{p}_1 + \mu\boldsymbol{p}_2 \qquad (3)$$

При $\mu \in [0,1]$ получаем точки прямой, лежащие между $\boldsymbol{p}_1$ и $\boldsymbol{p}_2$. При $\mu < 0$ – точки лежащие на прямой за $\boldsymbol{p}_1$, при $\mu > 1$ – точки, лежащие на прямой за $\boldsymbol{p}_2$. Для проверки этого просто подставьте в уравнение вместо μ значения 0 и 1.

Перейдем теперь к задаче вывода уравнения плоскости. Мы рассмотрим три способа его получения. Для этого прежде напомним определение скалярного произведения. Для двух радиус-векторов $\boldsymbol{p}$ и $\boldsymbol{q}$ скалярным произведением называется число $\boldsymbol{p} \cdot \boldsymbol{q} = |\boldsymbol{p}||\boldsymbol{q}|Cos\alpha$, где α – угол между векторами $\boldsymbol{p}$ и $\boldsymbol{q}$. Для векторов запись вида $\boldsymbol{pq}$ или $(\boldsymbol{p},\boldsymbol{q})$ также считается скалярным произведением. С практической точки зрения это определение может вызвать некоторое смущение. Действительно, вычислить угол между векторами, которые заданы координатами в трехмерном пространстве, вряд ли может показаться простым делом. Но, во-первых, часто бывает достаточно знать не само значение угла, а значение его косинуса, а во-вторых, скалярное произведение *в ортонормированной системе координат* можно выразить через координаты векторов:

$$\boldsymbol{p} \cdot \boldsymbol{q} = (p_1\boldsymbol{i} + p_2\boldsymbol{j} + p_3\boldsymbol{k})(q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k}) = p_1q_1 + p_2q_2 + p_3q_3 ,$$

так как при раскрытии скобок скалярные произведения перпендикулярных векторов базиса, по определению скалярного произведения, обращаются в ноль.

Уравнение плоскости

Используем свойства скалярного произведения для получения уравнения плоскости. Рассмотрим некоторую плоскость в пространстве и некоторую точку $\boldsymbol{a}$, про которую известно, что она лежит в этой плоскости, как показано на рисунке 5.

Возьмем также некоторый радиус-вектор $\boldsymbol{n}$, перпендикулярный нашей плоскости. Этот вектор назовем нормалью к плоскости. Пусть теперь требуется определить, принадлежит ли некоторая точка (или радиус-вектор) $\boldsymbol{p}$ плоскости или нет. Для этого заметим, что для любой точки $\boldsymbol{p}$, принадлежащей плоскости, вектор $(\boldsymbol{p}-\boldsymbol{a})$ и радиус-вектор нормали $\boldsymbol{n}$ – перпендикулярны. А это значит, что их скалярное произведение равно нулю:

$$\boldsymbol{n}(\boldsymbol{p}-\boldsymbol{a})=0 \qquad (4)$$

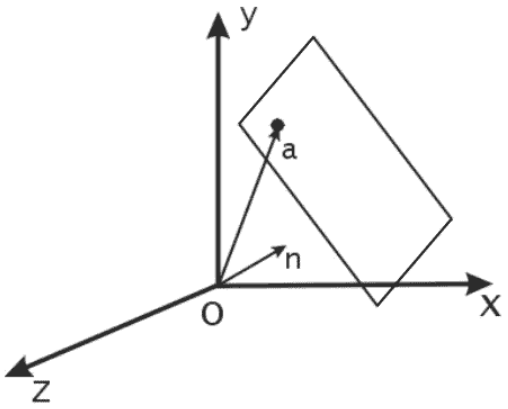

Рис. 5. Вывод уравнения плоскости в трехмерном пространстве.

Так, равенство (4) уже представляет собой уравнение плоскости в векторной форме. Раскрыв скобки, можно записать его в более удобном виде: $\boldsymbol{np}=c$, где константа $c=\boldsymbol{na}$. Если $\boldsymbol{n}=(A,B,C)$, а $\boldsymbol{p}=(x,y,z)$, то в координатной записи уравнение плоскости запишется в виде

$$Ax+By+Cz=c \qquad (5)$$

Рассмотрим далее второй способ получения уравнения плоскости, которая задана тремя неколлинеарными векторами, или тремя, не лежащими на одной прямой, точками. Для этого рассмотрим определение операции векторного произведения. Результатом векторного произведения двух векторов $\boldsymbol{p}\times\boldsymbol{q}$ является вектор $\boldsymbol{r}$, модуль которого равен $|\boldsymbol{p}\times\boldsymbol{q}|=|\boldsymbol{p}||\boldsymbol{q}|Sin\alpha$, и направлен он перпендикулярно плоскости, в которой лежат векторы $\boldsymbol{p}$ и $\boldsymbol{q}$, причем векторы $\boldsymbol{p,q,r}$ – образуют правую тройку векторов (см. определение правой системы координат), здесь α – угол между векторами $\boldsymbol{p}$ и $\boldsymbol{q}$. Для векторов единичного базиса, образующих правую тройку, как следует из определения: $\boldsymbol{i}\times\boldsymbol{j}=\boldsymbol{k}$, $\boldsymbol{j}\times\boldsymbol{k}=\boldsymbol{i}$, $\boldsymbol{k}\times\boldsymbol{i}=\boldsymbol{j}$. Векторное произведение так же подчиняется дистрибутивному закону, как и скалярное произведение. Однако векторное произведение не коммутативно, а именно, если для векторов $\boldsymbol{u}\times\boldsymbol{v}=\boldsymbol{w}$, то $\boldsymbol{v}\times\boldsymbol{u}=-\boldsymbol{w}$, что также прямо следует из его определения. Координаты векторного произведения можно получить, если разложить векторы, участвующие в произведении, по базису, а затем раскрыть скобки, подобно тому, как это уже было проделано для скалярного

произведения. Есть и другой, неформальный, но легче запоминаемый способ получения координат векторного произведения, с помощью разложения следующего определителя по его первой строке. Если $\boldsymbol{p} = (p_x, p_y, p_z)$ и $\boldsymbol{q} = (q_x, q_y, q_z)$, тогда

$$\boldsymbol{p} \times \boldsymbol{q} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ p_x & p_y & p_z \\ q_x & q_y & q_z \end{vmatrix} = (p_y q_z - p_z q_y) \cdot \boldsymbol{i} + (p_z q_x - p_x q_z) \cdot \boldsymbol{j} + (p_x q_y - p_y q_x) \cdot \boldsymbol{k}$$

Сведем теперь условия в новой постановке задачи нахождения уравнения плоскости к предыдущему случаю, где мы использовали вектор нормали. Пусть заданы фиксированные векторы $\boldsymbol{p}$, $\boldsymbol{q}$ и $\boldsymbol{r}$, не лежащие на одной прямой, определяющие плоскость, уравнение которой требуется получить (рис. 6).

Результат векторного произведения любых двух неколлинеарных векторов, параллельных нашей плоскости, будет вектором, перпендикулярным плоскости. И как

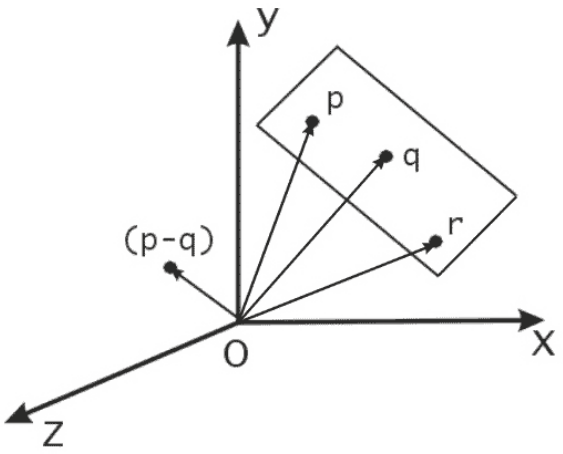

Рис. 6. Вывод уравнения плоскости, проходящей через три точки.

раз такими являются векторы разности $(\boldsymbol{p} - \boldsymbol{q})$ и $(\boldsymbol{p} - \boldsymbol{r})$. Выберем их векторное произведение в качестве вектора нормали, то есть $\boldsymbol{n} = (\boldsymbol{p} - \boldsymbol{q}) \times (\boldsymbol{p} - \boldsymbol{r})$. Тогда, если $\boldsymbol{x}$ – произвольный радиус-вектор, принадлежащий плоскости, то искомым уравнением плоскости будет, аналогично формуле (4):

$$[(\boldsymbol{p} - \boldsymbol{q}) \times (\boldsymbol{p} - \boldsymbol{r})] \cdot (\boldsymbol{x} - \boldsymbol{q}) = \boldsymbol{0},$$

причем уравнение этой же плоскости можно было бы записать, если в последней скобке вместо вектора $\boldsymbol{q}$ использовать векторы $\boldsymbol{p}$ или $\boldsymbol{r}$. Не будем далее расписывать это уравнение через координаты, так как это не трудно проделать самостоятельно.

Рассмотрим еще несколько определений и типичных задач, решение которых не должно вызывать затруднений при решении более сложных задач.

Некоторые элементарные задачи

Иногда бывает необходимо вычислить длину проекции радиус-вектора не на ось системы координат, а на другой радиус-вектор. Найдем длину проекции вектора $\boldsymbol{a}$ на вектор $\boldsymbol{b}$. Эта ситуация изображена на рис. 7, из которого, очевидно, следует и решение задачи.

Искомая проекция: $\mu = |\boldsymbol{a}| Cos\alpha = |\boldsymbol{a}| \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{|\boldsymbol{a}||\boldsymbol{b}|} = \boldsymbol{a} \cdot \frac{\boldsymbol{b}}{|\boldsymbol{b}|}$.

В случае тупого угла между векторами $\boldsymbol{a}$ и $\boldsymbol{b}$ значение μ в данном выражении будет отрицательным. Поэтому для получения длины проекции следует взять $|\mu|$.

Как видно, если длина вектора, на который проецируется другой вектор, равна единице, то длина проекции будет просто равна скалярному произведению этих векторов.

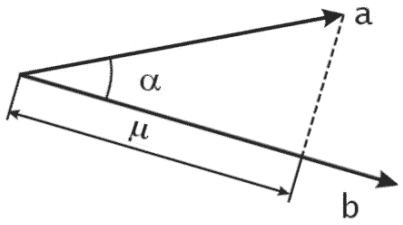

Рис. 7. Проекция вектора a на вектор b.

С помощью формулы длины проекции вектора на вектор можно еще одним способом получить уравнение плоскости, если заметить, что длины проекций радиус-векторов, принадлежащих плоскости, на вектор нормали к плоскости всегда равны между собой.

Рассмотрим задачу нахождения минимального расстояния от начала координат до плоскости. Очевидно, что это расстояние необходимо откладывать вдоль прямой, определяемой вектором нормали к плоскости. Но для нахождения этого расстояния надо найти сначала точку пересечения прямой с плоскостью. Поэтому найдем вначале решение задачи нахождения точки пересечения прямой и плоскости. Обозначим искомую точку, или соответствующий ей радиус-вектор $\boldsymbol{x}$. Тогда эта точка должна одновременно удовлетворять уравнениям прямой и плоскости, например, $\boldsymbol{x} = \boldsymbol{p}_1 + \mu \boldsymbol{p}^*$ *и* $\boldsymbol{n}\boldsymbol{x} = c$, где $\boldsymbol{p}_1$ и $\boldsymbol{p}^*$ – базовый и направляющий векторы, а $\boldsymbol{n}$ – вектор нормали к плоскости. Подставив $\boldsymbol{x}$ из первого уравнения во второе, найдем значение константы μ, которое затем подставим в исходное уравнение прямой для получения координат искомой точки:

$$\boldsymbol{n}(\boldsymbol{p}_1 + \mu \boldsymbol{p}^*) = c \Rightarrow \quad \boldsymbol{n}\boldsymbol{p}_1 + \mu \boldsymbol{n}\boldsymbol{p}^* = c \;\Rightarrow \mu = \frac{c - \boldsymbol{n}\boldsymbol{p}_1}{\boldsymbol{n}\boldsymbol{p}^*}$$

Уравнение прямой вдоль вектора нормали к плоскости запишем как $\boldsymbol{x} = \mu \cdot \boldsymbol{p}^*$, так как прямая проходит через начало координат и базовый вектор $\boldsymbol{p}_1$ равен нулю. Перед тем как подставить в это уравнение выражение для μ, заметим, что направляющий вектор совпадает с вектором нормали $\boldsymbol{p}^* = \boldsymbol{n}$. Учитывая это, запишем:

$$\boldsymbol{x} = \frac{c}{\boldsymbol{n} \cdot \boldsymbol{n}} \cdot \boldsymbol{n} = \frac{c}{|\boldsymbol{n}|^2} \cdot \boldsymbol{n}$$

Отсюда искомое расстояние от начала координат до плоскости равно

$$|\boldsymbol{x}| = \frac{c}{|\boldsymbol{n}|}$$

В том случае, когда вектор нормали $\boldsymbol{n}$ является нормированным, т.е. его длина равна 1,

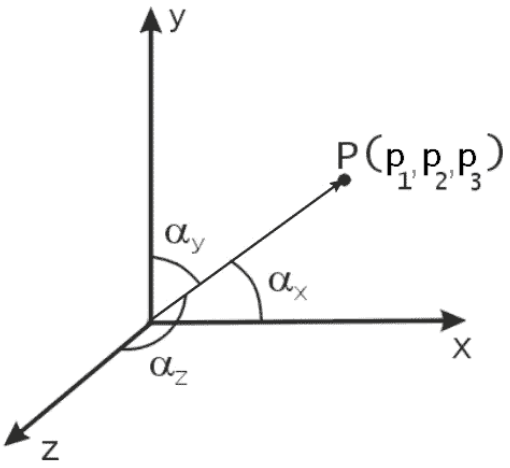

Рис. 8. Направляющие косинусы.

константа c в уравнении плоскости равна расстоянию от начала координат до данной плоскости.

Кроме определения положения точек в пространстве радиус-векторы также определяют направление в пространстве. Направление, определяемое радиус-вектором, удобно описывать с помощью, так называемых, направляющих косинусов. Пусть радиус-вектор $\boldsymbol{p} = (p_1, p_2, p_3)$ составляет с осями координат Ox, Oy и Oz углы, соответственно, α_x, α_y и α_z (рисунок 8). Тогда его направляющие косинусы равны:

$$Cos\,\alpha_x = \frac{p_1}{|\mathbf{p}|},\ Cos\,\alpha_y = \frac{p_2}{|\mathbf{p}|},\ Cos\,\alpha_z = \frac{p_3}{|\mathbf{p}|}$$

Отсюда, очевидно, вытекают следующие свойства направляющих косинусов:

$$Cos^2\alpha_x + Cos^2\alpha_y + Cos^2\alpha_z = 1.$$

Направляющие косинусы пропорциональны соответствующим координатам:

$$Cos\alpha_x : Cos\alpha_y : Cos\alpha_z = p_1 : p_2 : p_3\ ,$$

а в случае, когда вектор $\boldsymbol{p}$ нормирован, значения его координат равны соответствующим направляющим косинусам.

Уравнение плоскости можно представить как функцию трех переменных. Для этого в уравнении (5) перенесем константу из правой части в левую и запишем функцию трех переменных $f(x, y, z) = Ax + By + Cz - c$. Если подставить координаты точки, принадлежащей данной плоскости, в это уравнение, то $f(x, y, z) = 0$. Если же точка не принадлежит плоскости, то значение функции, очевидно, будет больше или меньше нуля. Интересен тот факт, что для точек, лежащих по одну и ту же сторону от плоскости, функция $f(x, y, z)$ имеет всегда один и тот же знак. С помощью чертежа на рисунке 9 , можно показать, что для точек, лежащих в полупространстве, порождаемом

плоскостью и содержащем начало координат, функция $f(x,y,z)$ отрицательна, а для точек, лежащих в другом полупространстве, как, например, для точки ***a*** на рисунке, она положительна. В общем же случае необходимо учитывать направление вектора нормали.

Свойство сохранения знака функции $f(x,y,z)$ удобно использовать в алгоритмах удаления невидимых ребер и граней для определения того, лежат ли точки по одну сторону от плоской грани или нет. Для этого достаточно лишь подставить значения координат точек в функциональное представление плоскости, определяемой соответствующей гранью, и проверить, совпадают знаки функции или нет. Аналогичные рассуждения можно провести и для более простого случая прямой на плоскости. Тогда для любой точки на плоскости можно определить ее нахождение в одной из полуплоскостей, на которые прямая делит плоскость. Это свойство используется в следующем примере.

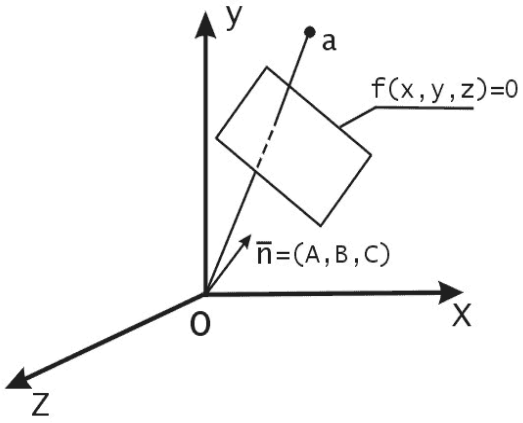

Рис. 9. Представление плоскости как функции трех переменных.

Рассмотрим методы решения классической задачи определения принадлежности точки внутренней или граничной области треугольника на плоскости. Эта задача имеет, конечно, много решений, некоторые из которых может придумать и сам читатель. Здесь приводится четыре из более чем двадцати методов решения этой задачи, известных автору. Лучшим из них может считаться метод, допускающий самую быструю программную реализацию, и в первую очередь это относится к минимизации количества операций умножения и деления, не говоря уже о трансцендентных функциях, вроде квадратного корня или тригонометрических функций, для вычисления которых требуется разложение в ряд. В начале рассмотрим один из известных методов решения этой задачи.

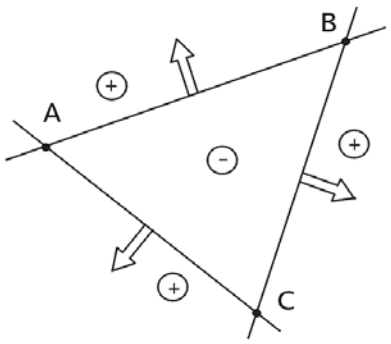

Рис. 10. Внутренняя область треугольника соответствует отрицательным направлениям векторов нормалей.

Пусть на плоскости xOy заданы три точки $A=(A_x,A_y), B=(B_x,B_y)$ и $C=(C_x,C_y),$ образующие треугольник (рис. 10). Через каждую пару вершин треугольника можно провести прямую. Ограниченная часть плоскости, образованная этими прямыми, есть внутренняя область треугольника. Если вектор нормали к прямой $\boldsymbol{n}=(L,M)$, то можно записать уравнение прямой на плоскости в виде: $Lx+My+N=0$. По знаку функции $f(x,y)=Lx+My+N$ можно определить нахождение произвольной точки с координатами (x,y) в той или иной полуплоскости относительно данной прямой. Идея первого метода состоит в том, чтобы записать

функциональные представления уравнений прямых, образующих стороны треугольника, таким образом, чтобы внутренняя область треугольника соответствовала, например, отрицательным значениям. Тогда условием принадлежности внутренней области треугольника будут отрицательные значения трех функциональных уравнений

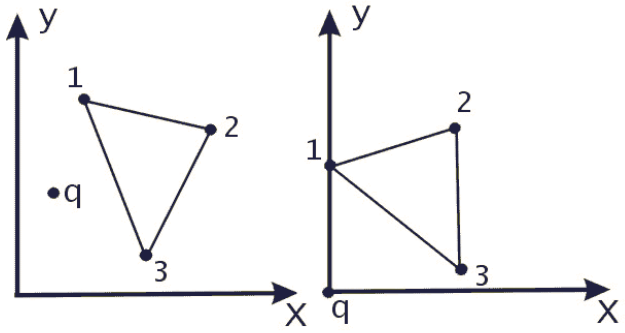

Рис. 11. Точка q вне треугольника.

прямых при подстановке координат проверяемой точки. Основной проблемой в этом методе является правильный выбор направления вектора нормали к прямой.

Следующий метод, разработанный автором в 1991 году, основан на преобразовании треугольника с помощью операции переноса таким образом, чтобы проверяемая точка совпала с началом координат. Поворотом плоскости вокруг начала координат расположим одну (любую) из вершин треугольника на оси Oy. Тогда если знаки координат x оставшихся двух точек совпадают, то искомая точка лежит вне треугольника. Если же знаки различны, то берем следующую из оставшихся вершин треугольника и поворотом плоскости устанавливаем ее на ось Oy. После чего вновь

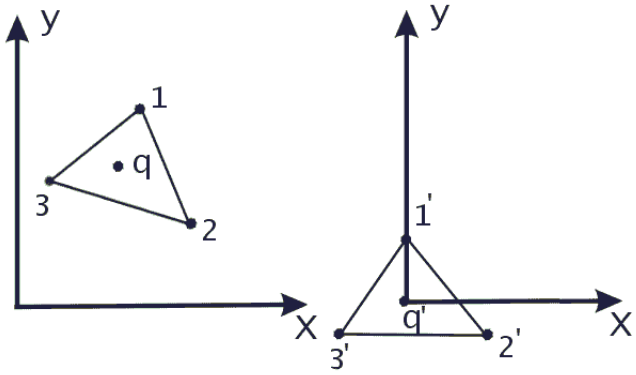

Рис. 2. Точка q внутри треугольника.

проверяем знаки координат x двух других вершин, и т.д.

Условием принадлежности точки внутренней области треугольника будет несовпадение знаков x - координат оставшихся двух вершин после каждого из трех поворотов.

Нахождение точки на одной из сторон треугольника легко определяется по несовпадению знаков y-координат двух вершин, которые после одного из поворотов оказались лежащими на оси Oy. Этот метод эффективен, когда больше вероятность, что точка лежит вне треугольника. Отрицательной его чертой является необходимость вычисления синусов и косинусов углов при повороте системы координат.

Третий из приводимых здесь методов до 2005 года считался мною одним из наиболее компактных и скоростных с вычислительной точки зрения. Этот метод был предложен автору Д. Чистяковым в 1999 году. Заметим, что очень просто можно определить принадлежность точки внутренней области треугольника – единичного симплекса, то есть треугольника, образованного точками с координатами $P=(0,0)$, $Q=(1,0)$, $R=(0,1)$. Для этого достаточно чтобы координаты искомой точки имели значения в отрезке $(0,1)$, и выполнялось условие $x+y<1$, где x и y - координаты точки. Заметим также, что с помощью аффинных преобразований и операций переноса или непрерывных деформаций любой треугольник можно преобразовать к единичному симплексу.

После таких преобразований внутренняя и внешняя области треугольника остаются таковыми. Применив такое преобразование к искомой точке, достаточно затем будет

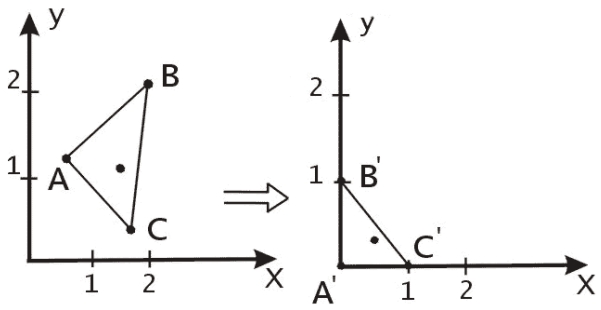

Рис. 3. Приведение произвольного треугольника к единичному симплексу.

определить ее нахождение во внутренней или внешней области симплекса. Найдем такое преобразование. Координаты векторов единичного базиса совпадают с координатами точек Q и R симплекса, соответственно. Будем считать, что точка C треугольника совпадает с началом координат. Этого всегда можно добиться параллельным переносом треугольника на вектор $-\overline{C}$. При этом координаты точек A и B треугольника суть коэффициенты разложения соответствующих векторов $\overline{A}$ и $\overline{B}$ по единичному базису. Матрица преобразования M от единичного базиса к базису на векторах $\overline{A}$ и $\overline{B}$ составлена из координат этих векторов.

$$M=\begin{bmatrix} A_x & A_y \\ B_x & B_y \end{bmatrix}$$

Значит, для обратного перехода к единичному базису, (на векторах которого построен симплекс), необходимо найти обратную матрицу:

$$M^{-1}=\frac{1}{A_xB_y-A_yB_x}\begin{bmatrix} B_y & -A_y \\ -B_x & A_x \end{bmatrix}.$$

Умножение радиус-вектора искомой точки на матрицу M^{-1} дает точку, которую достаточно проверить на попадание во внутреннюю или внешнюю область единичного симплекса, как было указано выше.

Идея четвертого метода, без строгого математического доказательства, была предложена одним из моих студентов Д. Трагером в 2005 году. Этот метод немного похож на метод углов приведенный в [8] и превосходит по скорости выполнения все из

выше упомянутых методов. После описания для него будет приведен текст процедуры на Delphi, написанный автором данной работы.

Зададим порядок обхода вершин и перенесем вершины треугольника и проверяемую точку так чтобы начало координат совпало с точкой *A*, например, как показано на рис. 14*a*. Конкретное направление обхода вершин треугольника, по или против часовой стрелки, не имеет значения. Рассмотрим углы α_1 и α_2, как показано на Рис. 14*a*, *b* и *c*, где *X* – точка, проверяемая на принадлежность внутренней области треугольника.

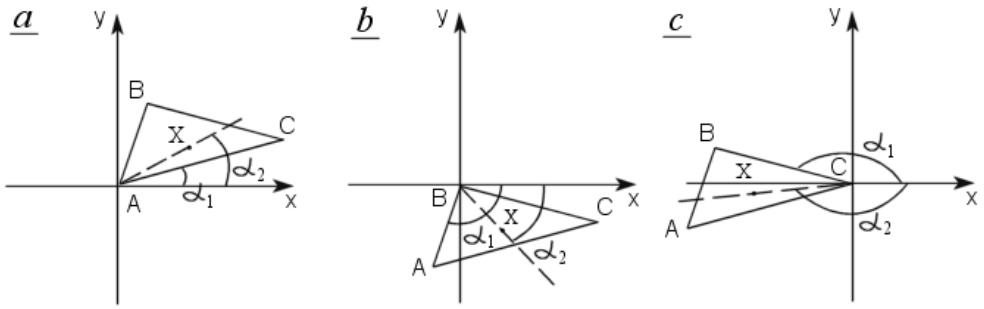

Рисунок 14. Последовательное измерение углов для получения разности $(\alpha_2 - \alpha_1)$.

Утверждается, что только в том случае когда точка *X* находится внутри треугольника, знак $Sin(\alpha_2 - \alpha_1)$ будет всегда совпадать для всех циклически выбираемых пар углов α_1 и α_2 при последовательном совмещении вершин треугольника с началом координат. Пусть вершинам треугольника *A*, *B, C* и проверяемой точке *X* соответствуют радиус-векторы $\boldsymbol{a} = (a_x, a_y)$, $\boldsymbol{b} = (b_x, b_y)$, $\boldsymbol{c} = (c_x, c_y)$ и $\boldsymbol{x} = (x, y)$.

Найдем выражение, по которому можно легко вычислить знак $Sin(\alpha_2 - \alpha_1)$.

Для ситуации, изображенной на Рис.14 *a*

$$Sin(\alpha_2 - \alpha_1) = Sin\alpha_2 Cos\alpha_1 - Cos\alpha_2 Sin\alpha_1 = \frac{(y - a_y)(c_x - a_x) - (x - a_x)(c_y - a_y)}{|\boldsymbol{x} - \boldsymbol{a}||\boldsymbol{c} - \boldsymbol{a}|}$$

Очевидно, что знак $Sin(\alpha_2 - \alpha_1)$ совпадает со знаком числителя правой части. Таким образом, получаем, что знак левой части определяется выражением

$$Sign((y - a_y)(c_x - a_x) - (x - a_x)(c_y - a_y)),$$

где *Sign* – функция получения знака числа. Аналогичные рассуждения для оставшихся двух случаев, изображенных на рисунках b и c, приводят, соответственно, к выражениям

$$Sign((y - b_y)(a_x - b_x) - (x - b_x)(a_y - b_y))$$

$$Sign((y - c_y)(b_x - c_x) - (x - c_x)(b_y - c_y)).$$

Доказательство правильности утверждения данного метода можно получить, если заметить, что знак $Sin(\alpha_2 - \alpha_1)$ не меняется при повороте плоскости относительно начала координат. Тогда, для каждого из трех случаев, изображенных на рисунках 14 *a*, *b* и *c*, соответствующими поворотами плоскости можно добиться того, чтобы угол α_1 был равен нулю, а оставшаяся часть треугольника, вместе с проверяемой точкой,

оказалась бы выше или ниже оси *ox,* в зависимости от выбранного направления обхода вершин треугольника.

Рассмотрим версию функции `InTri`, в которой реализуется данный метод для целочисленных координат треугольника *A*=(AX,AY), *B*=(BX,BY), *C*=(CX,CY) и проверяемой точки *X*=(x,y). Функция выдает результат TRUE, если точка находится внутри или на границе треугольника (граница, таким образом, считается принадлежащей внутренней области треугольника), или FALSE, если точка находится за его пределами. Этой функцией удобно пользоваться если точки находятся на растровой сетке монитора, где координаты являются целыми числами.

```
function InTriI(x,y,AX,AY,BX,BY,CX,CY: Integer): boolean;
var
   Res1,Res2: Integer;
begin
 Res1:= (y-ay)*(cx-ax)-(x-ax)*(cy-ay);
 Res2:= (y-cy)*(bx-cx)-(x-cx)*(by-cy);
 // if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
   if Res1 XOR Res2 < 0 then
    Result:= false //дальше не вычисляем
  else
   begin
    Res2:= (y-by)*(ax-bx)-(x-bx)*(ay-by);
    // if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
    if Res1 XOR Res2 < 0 then
       Result:= false
     else
       Result:= true;
   end;
end;
```

Обратите внимание на пару строк, которые дважды встречаются в тексте функции.

```
// if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
   if Res1 XOR Res2 < 0 then
```

Первая из двух строк специально закомментирована, так как вторая представляет собой ее оптимизированный вариант. Изначально в методе требуются действия, записанные в первой строке, где производится проверка на совпадение или несовпадение знаков вычисленных углов, что определяется переменными Res1 и Res2. Поскольку знак целого числа определяется его левым битом, точнее число считается отрицательным, если самый старший его бит равен 1, то он всегда будет равен единице при умножении двух чисел разного знака. Операцией целочисленного умножения здесь можно было бы воспользоваться, если бы операция XOR не давала аналогичный результат значительно быстрее. Код ассемблера, который генерирует Delpi, в случае использования операции XOR, примерно в 4 раза короче чем тот, что получается при использовании первой из двух рассмотренных нами строк на Паскале. Точнее, при включенной опции оптимизации компиляции в Delphi, вторая строка реализуется всего двумя инструкциями ассемблера, включая команду условного перехода. Но самое интересное, что аналогичный программный трюк можно применить и для вещественных входных параметров функции – типов Single и Double. Я не буду более вдаваться в технические тонкости, просто посмотрите текст функций. В случае вещественных входных параметров в них появляются небольшие отличия от целочисленного варианта.
Версия функции `InTri` для входных параметров типа Single.

```
function InTriS(x,y,AX,AY,BX,BY,CX,CY: Single): boolean;
var
   Res1,Res2: Single;
   ReI1: Integer absolute Res1;
   ReI2: Integer absolute Res2;
begin
 Res1:= (y-ay)*(cx-ax)-(x-ax)*(cy-ay);
 Res2:= (y-cy)*(bx-cx)-(x-cx)*(by-cy);
 // if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
   if ReI1 XOR ReI2 < 0 then
    Result:= false //дальше не вычисляем
  else
   begin
    Res2:= (y-by)*(ax-bx)-(x-bx)*(ay-by);
    // if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
    if ReI1 XOR ReI2 < 0 then
       Result:= false
     else
       Result:= true;
   end;
end;//function InTriS
```

Версия функции `InTri` для входных параметров типа Double.

```
function InTriD(x,y,AX,AY,BX,BY,CX,CY: Double): boolean;
var
   Res1,Res2: Double;
   ReI1: Int64 absolute Res1;
   ReI2: Int64 absolute Res2;
begin
 Res1:= (y-ay)*(cx-ax)-(x-ax)*(cy-ay);
 Res2:= (y-cy)*(bx-cx)-(x-cx)*(by-cy);
 // if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
   if ReI1 XOR ReI2 < 0 then
    Result:= false //дальше не вычисляем
  else
   begin
    Res2:= (y-by)*(ax-bx)-(x-bx)*(ay-by);
    // if (Res1>0)and(Res2<0)or(Res1<0)and(Res2>0) then
    if ReI1 XOR ReI2 < 0 then
       Result:= false
     else
       Result:= true;
   end;
end;//function InTriD
```

В результате, как можно видеть по тексту приведенных процедур, задача решается в худшем случае за 6 операций умножения и 15 операций вычитания, а в лучшем случае всего за 4 операции умножения и 10 операций вычитания.

Глава 2. Проецирование трехмерных объектов

Классификация проекций

Для того, чтобы увидеть на плоскости монитора трехмерное изображение, нужно уметь задать способ отображения трехмерных точек в двумерные. В общем случае проекции преобразуют точки, заданные в системе координат размерностью n в точки системы координат размерностью меньшей, чем n. В нашем случае точки трехмерного пространства преобразуются в точки двумерного пространства.

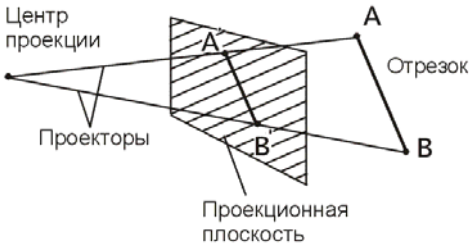

Рис. 4. Центральная проекция.

Чтобы построить проекцию нужно задать точку, которая называется центром проекции. Проекции строятся с помощью проецирующих лучей или проекторов, которые выходят из центра проекции. Проекторы пересекают плоскость, которая называется проекционной или картинной плоскостью, и затем проходят через каждую точку трехмерного объекта и образуют тем самым проекцию (см. Рис. 15). Из всего множества типов проекций будем рассматривать лишь плоские геометрические проекции, которые получаются на плоскости, а не на искривленной поверхности, и в качестве проекторов используются прямые, а не искривленные линии. Плоские геометрические проекции делятся на два вида: центральные и параллельные. Если центр проекции находится на конечном расстоянии от проекционной плоскости, то проекция – центральная. Если же центр проекции удален на бесконечность, то проекция – параллельная.

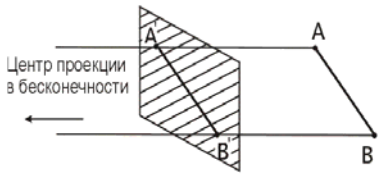

Рис. 5 Параллельная проекция.

Одним из интересных свойств центральной проекции являются так называемые точки схода. Точка схода есть точка пересечения центральных проекций любой совокупности параллельный прямых, которые не параллельны проекционной плоскости. Существует бесконечное множество точек схода. Точка схода называется главной, если совокупность прямых параллельна одной из координатных осей. В зависимости от того, сколько координатных осей пересекает проекционную плоскость, различают одно-, двух- и трехточечные проекции.

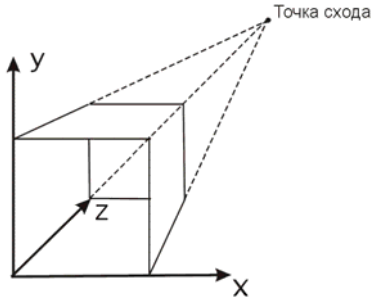

Рис. 6. Одноточечная проекция.

Перечислим основные виды проекций, которые используются во многих приложениях компьютерной графики.

Простейшей является параллельная прямоугольная проекция. В ней совместно изображаются виды сверху, спереди и сбоку. Эти проекции часто используются в при создании чертежей трехмерных объектов. В зависимости от соотношения между

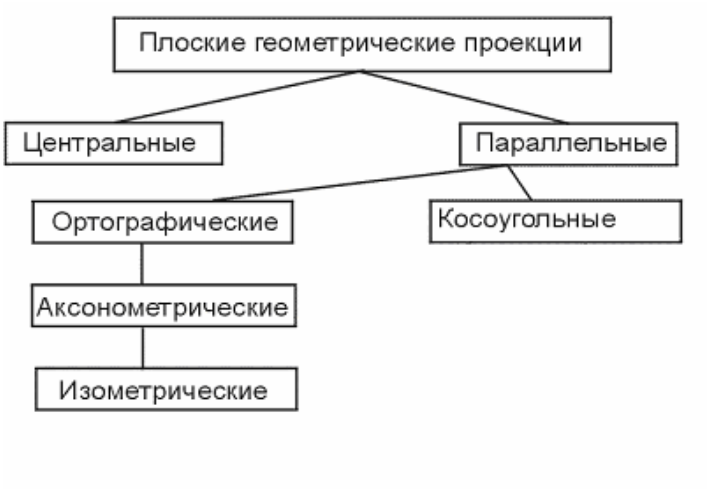

Рис. 7. Типы проекций.

направлениями проецирования и нормалью к проекционной плоскости параллельные проекции разделяются на ортографические или ортогональные, в которых эти направления совпадают, и косоугольные, в которых они не совпадают. В зависимости от положения осей системы координат объекта относительно проекционной плоскости ортографические проекции делятся на аксонометрические и изометрические. В изометрических проекциях оси системы координат составляют одинаковые углы с проекционной плоскостью. В аксонометрических проекциях эти углы разные.

Центральная перспективная проекция приводит к визуальному эффекту, подобному тому, который дает зрительная система человека. При этом наблюдается эффект перспективного укорачивания, когда размер проекции объекта изменяется обратно пропорционально расстоянию от центра проекции до объекта. В параллельных проекциях отсутствует перспективное укорачивание, за счет чего изображение получается менее реалистичным, и параллельные прямые всегда остаются параллельными.

Вывод формул центральной перспективной проекции

Для получения формул центральной перспективной проекции расположим оси системы координат, проекционную плоскость и центр проекции как показано на рис. 19.

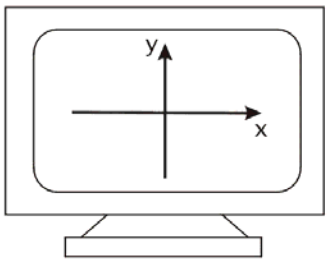

Рис. 8. Расположение осей координат на экране.

Будем имитировать на экране то, что как будто бы реально находится в пространстве за ним. Заметим, что система координат, изображенная на рис. 19 – левосторонняя. Будем считать, что плоскость экрана монитора совпадает с проекционной плоскостью. Прежде чем переходить к собственно вычислениям следует сделать одно важное замечание. Поскольку поверхность любого трехмерного объекта содержит бесконечное число точек, то необходимо задать способ описания

поверхности объекта конечным числом точек для представления в компьютере. А именно, будем использовать линейное представление объектов в трехмерном пространстве с помощью отрезков прямых и плоских многоугольников. При этом отрезки прямых после перспективного преобразования переходят в отрезки прямых на проекционной плоскости. Доказательство этого достаточно простое и здесь не приводится. Это важное свойство центральной перспективы позволяет проецировать, т.е. производить вычисления только для конечных точек отрезков, а затем соединять проекции точек линиями уже на проекционной плоскости.

Точка A проецируется на экран как A'. Расстояние от наблюдателя до проекционной

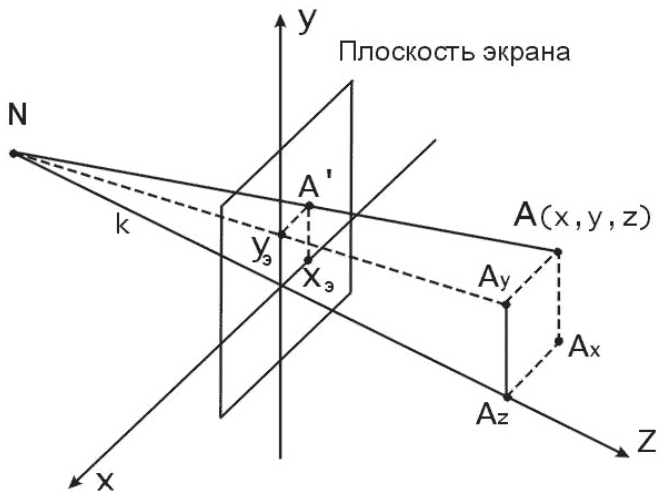

Рис. 20. Вывод формул центральной перспективной проекции.

плоскости равно k. Определим координаты точки A' на экране. Обозначим их $x_э$ и $y_э$. Из подобия треугольников $A_y A_z N$ и $y_э ON$ находим, что

$$\frac{y}{z+k}=\frac{y_э}{k},\Rightarrow y_э=\frac{ky}{z+k} \qquad (1)$$

аналогично для x: $x_э=\frac{kx}{z+k}$.

Напомним, что k -это расстояние, а наблюдатель находится в точке $N=(0,0,-k)$.

Если точку наблюдения поместить в начало координат, а проекционную плоскость на расстояние a, как показано на рис. 21, то формулы для $x_э$ и $y_э$ примут вид:

$$x_э=\frac{kx}{z},\qquad y_э=\frac{ky}{z} \qquad (2)$$

Формулы (1) более удобны при необходимости простым образом приближать или удалять наблюдателя от проекционной плоскости. Формулы (2) требуют меньше времени для вычислений за счет отсутствия операции сложения.

Рассмотрим далее некоторые факторы. влияющие на восприятие человеком трехмерности. Одним из простых способов представления трехмерных объектов являются так называемые проволочные изображения. Кривые линии при этом

апроксимируются отрезками прямых. Это наиболее быстрый и простой способ изображения.

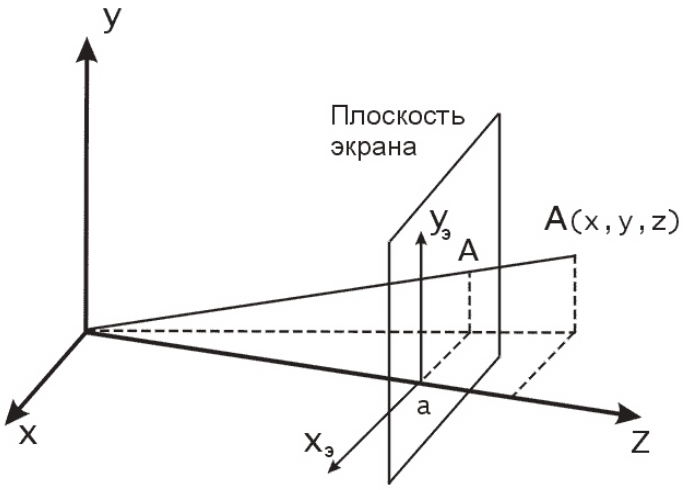

Рис. 21. Другой способ вычисления координат точек в центральной перспективной проекции.

Для усиления эффекта трехмерной глубины в проволочных изображениях объектов удаляют невидимые линии. Линии или их части, закрытые поверхностями объекта, не изображаются. Для этого применяется специальный алгоритм, что требует уже больших вычислений. Передача глубины может осуществляться изменением уровня яркости. Объекты, которые находятся ближе к наблюдателю, изображаются ярче, чем те, которые расположены дальше от него. Движение объектов также дает дополнительный эффект глубины. Например, вращение объектов вокруг вертикальной оси позволяет отличить точки, находящиеся на разном расстоянии от оси за счет различия линейной скорости вращения точек. Это так называемый кинетический или динамический эффект глубины.

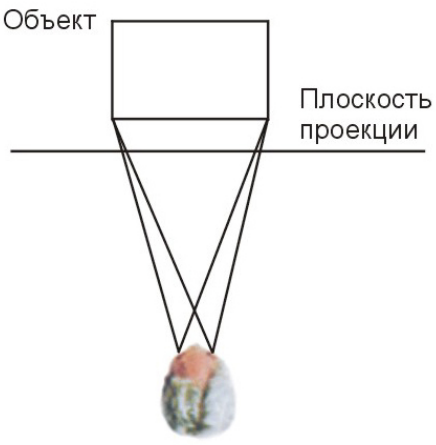

Рис. 22. Бинокулярный эффект, стереоскопия.

Более тонко трехмерность объектов может быть представлена за счет различий отражательных способностей поверхностей, их рельефа и текстуры, а также расчета теней, отбрасываемых поверхностями объекта. Одним из редко используемых, но наиболее эффективных способов достижения эффекта трехмерности является стереоскопия. При этом отдельно для правого и левого глаза наблюдателя формируются изображения, которые незначительно отличаются друг от друга, подобно тому, как это происходит в реальности. Это вызывает так называемый бинокулярный эффект, который заключается в том, что наш мозг сливает два отдельных образа в один, интерпретируемый как трехмерный. Эти два раздельных изображения называются стереопарой.

Технически этот метод реализуется, например, с помощью очков со специальными поляризованными стеклами. На экран монитора поочередно выводятся изображения для левого и правого глаза. А стекла очков становятся поочередно, соответственно, прозрачными или непрозрачными. При достаточно частой смене изображений смены состояний прозрачности и непрозрачности не ощущается. Поскольку при изменении положения головы центр проекции остается на месте, то создается псевдо-трехмерный эффект. Синхронизация смены кадров на экране и поляризации линз очков происходит с помощью специальных датчиков, расположенных на очках и мониторе.

Глава 3. Преобразования в пространстве

Преобразования точек в разных системах координат

Необходимо научиться управлять изображением на экране, вносить изменения в его положение, форму, ориентацию, размер. Для этих целей существует набор геометрических преобразований, которые позволяют изменять эти характеристики объектов в пространстве. Представим задачу создания компьютерного имитатора полетов на самолете. Будем считать, что перспективная проекция трехмерных объектов должна рассчитываться как вид из кабины пилота. Объекты на земле, которые видит пилот, изменяют свое положение. Например, автомобиль движется относительно земли, колеса автомобиля вращаются. При этом, наблюдатель видит эту картину из определенной точки в пространстве в выбранном направлении. Необходимо описать эти сложные преобразования математически.

Введем три вида систем координат. Первая из них – мировая система координат – задается осями $X_M Y_M Z_M$. Мы размещаем ее в некоторой точке, и она остается неподвижной всегда. Вторая – система координат наблюдателя. Эту систему назовем $X_N Y_N Z_N$. Она определяет положение наблюдателя в пространстве и задает направление взгляда. И третья – система координат объекта. В нашем случае их две: система координат автомобиля и система координат его колес. Эти системы могут перемещаться и изменять свое положение в пространстве относительно мировой системы координат. Координаты точек объектов задаются в системах координат объектов, каждая из которых, в свою очередь, привязана к мировой системе координат. Система координат наблюдателя также перемещается относительно мировой системы координат. Теперь становится понятно, что для того, чтобы увидеть трехмерный объект на экране компьютера надо проделать следующие шаги.

1. Преобразовать координаты объекта, заданные в собственной системе координат, в мировые координаты.
2. Преобразовать координаты объекта, заданные уже в мировой системе координат, в систему координат наблюдателя.
3. Спроецировать полученные координаты на проекционную плоскость в системе координат наблюдателя.

В библиотеках компьютерной графики, таких как OpenGL и Direct3D, приведенные здесь типы преобразований задаются в виде матриц и имеют собственные устоявшиеся названия. Первое преобразование называется мировым, соответствующая матрица мирового преобразования (WORLD MATRIX). Второе преобразование называется модельным или видовым и определяется матрицей модельного или, соответственно, видового преобразования (VIEW MATRIX). Третье преобразование называется преобразованием проекции и соответствующая матрица – матрицей проецирования (PROJECTION MATRIX).

Отметим, определенную двойственность, возникающую при взаимных перемещениях систем координат друг относительно друга. Представим себе, что мы наблюдаем кубик в пространстве. Пусть этот кубик начнет вращаться, например, вокруг вертикальной оси. Мы увидим, что кубик вращается. Но тот же самый эффект мы получим, если сами начнем облетать вокруг кубика и рассматривать его с разных сторон. Визуальный эффект остается тем же самым, хотя в первом случае наша система координат остается неподвижной, а во втором – вращается по орбите. Этот эффект можно использовать при выводе формул движения в пространстве.

Двумерные матричные преобразования

Рассмотрим три основных типа преобразований точек, которые часто используются в приложениях компьютерной графики. Это операции переноса, масштабирования и поворота или вращения. В данном параграфе будем рассмотривать преобразования координат точек на плоскости.

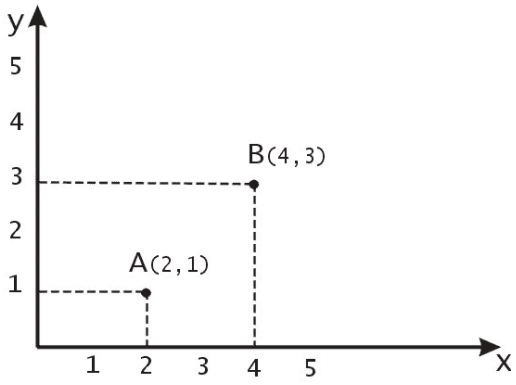

Рис. 9. Операция переноса или трансляции точки A в точку B.

На рис. 23 точка A перенесена в точку B. Математически этот перенос можно описать с помощью вектора переноса $\overline{AB}$. Пусть $\overline{R}$ радиус вектор, соответствующий вектору переноса $\overline{AB}$. Тогда переход из точки A в точку B будет соответствовать векторной записи $\overline{B} = \overline{A} + \overline{R}$. Отсюда получаем, что для переноса точки в новое положение необходимо добавить к ее координатам некоторые числа, которые представляют собой координаты вектора переноса:

$$\overline{B} = \overline{A} + \overline{R} = \left[A_x + R_x, A_y + R_y\right]$$

Масштабированием объектов называется растяжение объектов вдоль соответствующих осей координат относительно начала координат. Эта операция применяется к каждой точке объекта, поэтому можно также говорить о масштабировании точки. При этом, конечно, речь не идет об изменении размеров самой точки. Масштабирование достигается умножением координат точек на некоторые константы. В том случае, когда эти константы равны между собой, масштабирование называется однородным. На рис. 24 приведен пример однородного масштабирования треугольника ABC.

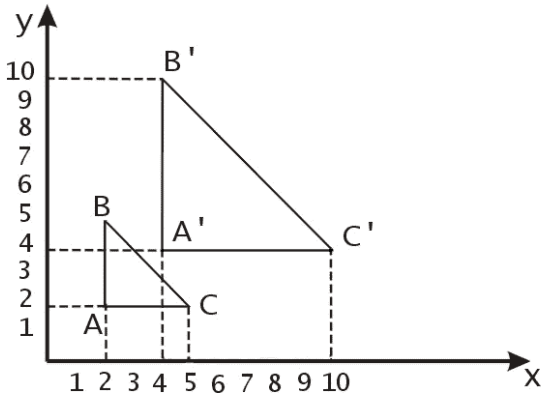

Рис. 10. Операция масштабирования .

После применения операции однородного масштабирования с коэффициентом 2 он переходит в треугольник $A'B'C'$. Если ввести матрицу масштабирования $S = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$, то для точек A и A' можно представить операцию масштабирования следующим образом:

$$\left[x', y'\right] = \left[x, y\right] \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}.$$

Рассмотрим далее операцию вращения точки на некоторый угол относительно начала координат. На рисунке 25 точка $A=(x,y)$ переходит в точку $B=(x',y')$ поворотом на угол α.

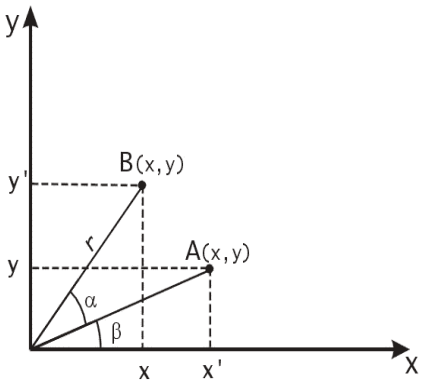

Рис. 11. Операция поворота (вращения) относительно начала координат.

Найдем преобразование координат точки A в точку B. Обозначим β угол, который составляет радиус-вектор $\overline{A}$ с осью O*x*. Пусть r – длина радиус-вектора $\overline{A}$, тогда

$$x' = r \cdot Cos(\alpha+\beta) = r(Cos\alpha \cdot Cos\beta - Sin\alpha \cdot Sin\beta)$$

$$y' = r \cdot Sin(\alpha+\beta) = r(Sin\alpha \cdot Cos\beta + Cos\alpha \cdot Sin\beta)$$

Так как $Cos\beta = x/r$ и $Sin\beta = y/r$, то подставляя эти выражения в уравнения для x' и y', получаем:

$$x' = x \cdot Cos\alpha - y \cdot Sin\alpha$$

$$y' = x \cdot Sin\alpha + y \cdot Cos\alpha$$

Поворот точки A на угол α относительно начала координат выглядит следующим образом:

$$[x', y'] = [x, y]\begin{bmatrix} Cos\alpha & Sin\alpha \\ -Sin\alpha & Cos\alpha \end{bmatrix}$$

Однородные координаты и матричное представление двумерных преобразований

В предыдущем параграфе были рассмотрены три вида преобразований точек на плоскости. Два из них – операции вращения и масштабирования - описываются в виде произведения матрицы на вектор, а третья – операция переноса – описывается как сумма двух векторов. В случае последовательного выполнения любой комбинации операций вращения и масштабирования результат легко можно записать в виде произведения матриц соответствующих преобразований. Это будет матрица результирующего поворота и масштабирования. Очевидно, что удобнее применять результирующую матрицу вместо того, чтобы каждый раз заново вычислять произведение матриц. Однако, таким способом нельзя получить результирующую матрицу преобразования, если среди последовательности преобразований присутствует хотя бы один перенос. Матричное произведение в компьютерной графике также называют композицией. Было бы удобнее иметь математический аппарат, позволяющий включать в композиции преобразований все три выше указанные операции. При этом получился бы значительный выигрыш в скорости вычислений. Введение однородных координат позволяет решить эту проблему*.

* Более строгое определение однородных координат дается в разделе линейной алгебры «Проективные пространства».

Так, двумерный вектор (x, y) в однородных координатах записывается в виде (wx, wy, w), где $w \neq 0$. Число w называется масштабным множителем. Для того, чтобы из вектора, записанного в однородных координатах получить вектор в обычных координатах необходимо разделить первые две координаты на третью: $(wx/w, wy/w, w/w) \to (x, y, 1)$.

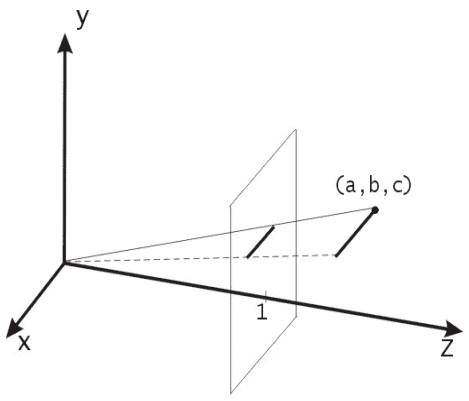

Рис. 12. Проекция на плоскость *z*=1.

В общем случае осуществляется переход от *n*-мерного пространства к $(n+1)$-мерному. Это преобразование не единственное. Обратное преобразование называется проекцией однородных координат.

Рассмотрим некоторые свойства однородных координат. Некоторые точки, неопределенные в n-мерном пространстве, становятся вполне определенными при переходе к однородным координатам. Например, однородный вектор $(0,0,1,0)$ в трехмерном пространстве соответствует бесконечно удаленной точке $z = \infty$. Поскольку в однородных координатах эту точку можно представить в виде $(0,0,1,\varepsilon)$, при $\varepsilon \to 0$, то в трехмерном пространстве это соответствует точке $(0,0,1/\varepsilon)$.

Рассмотрим точку трехмерного пространства (a, b, c). Если представить эту точку как однородное представление точки двумерного пространства, то ее координаты будут $(a/c, b/c)$. Сравнивая эти координаты со вторым видом формул, выведенных для центральной перспективной проекции, легко заметить, что двумерное представление точки с координатами (a, b, c) выглядит как ее проекция на плоскость $z = 1$, как показано на рис. 26.

Аналогично, рассматривая применение однородных координат для векторов трехмерного пространства, можно представить трехмерное пространство как проекцию четырехмерного пространства на гиперплоскость $w = 1$, если $(x, y, z) \to (wx, wy, wz, w) = (x, y, z, 1)$.

В однородных координатах преобразование центральной перспективы можно определить матричной операцией. Эта матрица записывается в виде:

$$\begin{bmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & k \end{bmatrix} = P$$

Покажем, что эта матрица определяет преобразование точки объекта, заданной в однородных координатах, в точку перспективной проекции (также в однородных координатах). Пусть $p = (x, y, z)$ – точка в трехмерном пространстве. Ее однородное представление $\mathrm{v} = (wx, wy, wz, w)$. Умножим v на P:

$$\mathrm{v}P = [wkx, wky, 0, w(z+k)] = [kx/(z+k), ky/(z+k), 0, 1]$$

- это в точности повторяет формулы (1), выведенные для центральной перспективы.

Теперь точки двумерного пространства будут описываться трехэлементными вектор-строками, поэтому и матрицы преобразований, на которые будет умножаться вектор точки, будут иметь размеры 3×3. Запишем матричное преобразование операции переноса для однородных координат:

$$\left[x', y', 1\right] = \left[x, y, 1\right] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x & D_y & 1 \end{bmatrix}$$

или $p' = p \cdot T\left(D_x, D_y\right)$, где $T\left(D_x, D_y\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x & D_y & 1 \end{bmatrix}$.

При последовательном переносе точки p в точку p' и затем в точку p'' компоненты суммарного вектора переноса являются суммами соответствующих компонент последовательных векторов переноса. Рассмотрим, каковы будут элементы матрицы суммарного переноса. Пусть $p' = p \cdot T\left(D_x, D_y\right)$, $p'' = p' \cdot T\left(D'_x, D'_y\right)$. Подставив первое уравнение во второе получаем $p'' = p \cdot T\left(D_x, D_y\right) \cdot T\left(D'_x, D'_y\right)$. Матричное произведение т.е. суммарный перенос равен произведению соответствующих матриц переноса.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x & D_y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D'_x & D'_y & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x + D'_x & D_y + D'_y & 1 \end{bmatrix}$$

Запишем матричный вид операции масштабирования.

$$\left[x', y', 1\right] = \left[x, y, 1\right] \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Определим матрицу масштабирования $S\left(S_x, S_y\right) = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Так же, как последовательные переносы являются аддитивными, покажем, что последовательные масштабирования будут мультипликативными.

$$S\left(S_x, S_y\right) \cdot S\left(S'_x, S'_y\right) = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S'_x & 0 & 0 \\ 0 & S'_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x \cdot S'_x & 0 & 0 \\ 0 & S_y \cdot S'_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Для операции поворота матричный вид будет такой:

$$\left[x', y', 1\right] = \left[x, y, 1\right] \cdot \begin{bmatrix} Cos\alpha & Sin\alpha & 0 \\ -Sin\alpha & Cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Определим матрицу поворота $R(\alpha) = \begin{bmatrix} Cos\alpha & Sin\alpha & 0 \\ -Sin\alpha & Cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Аналогично двум предыдущим случаям, покажем, что матрица поворота остается таковой при последовательных поворотах.

$$R(\alpha)R(\beta) = \begin{bmatrix} Cos\alpha & Sin\alpha & 0 \\ -Sin\alpha & Cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Cos\beta & Sin\beta & 0 \\ -Sin\beta & Cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} Cos(\alpha+\beta) & Sin(\alpha+\beta) & 0 \\ -Sin(\alpha+\beta) & Cos(\alpha+\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Таким образом, доказано, что два, а значит и любое количество последовательных поворотов можно записать в виде одной матрицы суммарного поворота. Также легко заметить что любая последовательность операций, включающая в себя перенос, масштабирование и вращение в однородных координатах, может быть представлена одной матрицей, которая является произведением матриц данных операций.

Рассмотрим, каким образом с помощью композиции матричных преобразований можно получить одно общее результирующее преобразование. Для этого будем использовать матрицы T, S и R. С вычислительной точки зрения гораздо проще и быстрее применять матрицу уже готового преобразования вместо того, чтобы применять их последовательно одну за другой. К точке более эффективно применять одно результирующее преобразование, чем ряд преобразований друг за другом.

Для примера рассмотрим задачу поворота объекта на плоскости относительно некоторой произвольной точки p_0. Пока мы умеем поворачивать объекты только вокруг начала координат. Но можно представить эту задачу как последовательность шагов, на каждом из которых будет применяться только элементарная операция: перенос, масштабирование или вращение.

Вот эта последовательность элементарных преобразований (рис. 27):

1. Перенос, при котором точка p_0 переходит в начало координат.
2. Поворот на заданный угол.

3. Перенос, при котором точка из начала координат возвращается в первоначальное положение p_0.

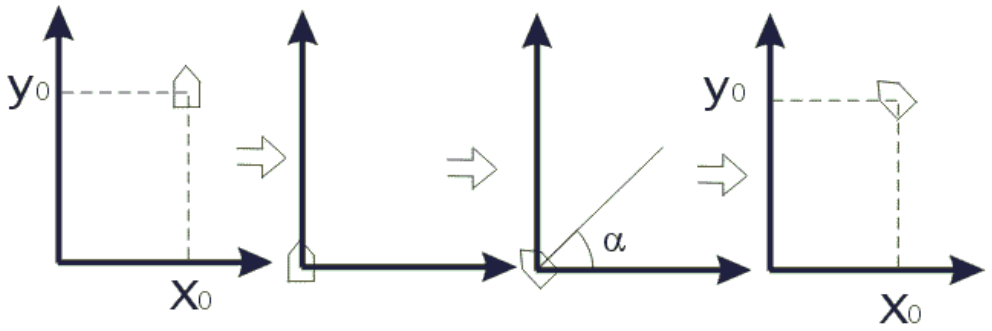

Рис. 13. Последовательность преобразований при повороте объекта вокруг точки $p_0 = (x_0, y_0)$ на угол α.

Точка $p_0 = (x_0, y_0)$. Первый перенос производится на вектор $[-x_0, -y_0]$, а обратный перенос - на вектор $[x_0, y_0]$.

Трехмерные матричные преобразования

Подобно тому, как двумерные преобразования описываются матрицами размером 3×3, трехмерные преобразования могут быть представлены матрицами размером 4×4. Тогда трехмерная точка (x, y, z) записывается в однородных координатах как (wx, wy, wz, w), где $w \neq 0$. Для получения декартовых координат надо первые три однородные координаты разделить на w. Два однородных вектора описывают одну декартову точку в трехмерном пространстве, если $H_1 = cH_2$, где $c = Const \neq 0$ и H_1, H_2 - векторы, записанные в однородных координатах.

Матрицы преобразований будем записывать в правосторонней системе координат. При этом положительный поворот определяется следующим образом. Если смотреть из положительной части оси вращения (например, оси z) в направлении начала координат, то поворот на 90° против часовой стрелки будет переводить одну положительную полуось в другую (ось x в y, в соответствии с правилом циклической перестановки).

Заметим, что на практике удобнее применять левостороннюю систему координат, так как в этом случае удобнее интерпретировать тот факт, что точки с большими значениями z находятся дальше от наблюдателя.

Запишем теперь матрицу трехмерного переноса. Аналогично двумерному случаю:

$$T(D_x, D_y, D_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{bmatrix}, \text{ при этом}$$

$$[x, y, z, 1] \cdot T(D_x, D_y, D_z) = [x + D_x, y + D_y, z + D_z, 1].$$

Операция масштабирования:

$$S(S_x, S_y, S_z) = \begin{bmatrix} S & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[x, y, z, 1] \cdot S(S_x, S_y, S_z) = [S_x \cdot x, S_y \cdot y, S_z \cdot z, 1]$$

Перейдем к операции поворота; с ней, в трехмерном случае, придется разбираться чуть побольше чем в двумерном. Так как при двумерном повороте в плоскости xy координаты z остаются неизменными, то поворот вокруг оси z записывается так:

$$R_z(\alpha) = \begin{bmatrix} Cos\alpha & Sin\alpha & 0 & 0 \\ -Sin\alpha & Cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матрица поворота вокруг оси x имеет вид:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & Cos\alpha & Sin\alpha & 0 \\ 0 & -Sin\alpha & Cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

и вокруг оси y:

$$R_y(\alpha) = \begin{bmatrix} Cos\alpha & 0 & -Sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ Sin\alpha & 0 & Cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Обратите внимание на смену положения синуса угла с отрицательным знаком в матрице поворота вокруг оси y. Правильность этих матриц легко проверить поворотом одного из ортов на 90°, при этом он должен перейти в следующий по порядку орт на соответствующей координатной оси.

Обратные преобразования будут выражаться обратными матрицами. Для операции переноса надо лишь заменить знаки компонент вектора переноса на противоположные:

$$T^{-1}(D_x, D_y, D_z) = T(-D_x, -D_y, -D_z);$$

для операции масштабирования – на обратные значения:

$$S^{-1}(S_x, S_y, S_z) = S(1/S_x, 1/S_y, 1/S_z);$$

для поворота – выбором отрицательного угла поворота:

$$R^{-1}(\alpha) = R(-\alpha).$$

Результатом нескольких последовательных поворотов будет матрица

$$A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Здесь верхняя матрица размером 3×3 называется ортогональной. Важным ее свойством является то, что обратная к ней матрица является транспонированной: $B^{-1} = B^T$. Это полезно тем, что при вычислениях достаточно поменять индексы местами и обратное преобразование получается автоматически.

После перемножения любого числа матриц вида T, S и R результирующая матрица всегда будет иметь вид:

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}.$$

Здесь верхняя часть размером 3×3 определяет суммарный поворот и масштабирование, а три коэффициента последней строки – суммарный перенос.

Вопросы эффективности вычислений

Рассмотрим проблему ускорения вычислений в одной из самых трудоемких операций компьютерной графики – операции поворота точки относительно начала координат. Как было показано ранее, для ее выполнения необходимо произвести 4 операции умножения, 2 операции сложения, а также вычислить значения синуса и косинуса угла поворота. Напомним вид формул поворота:

$$x' = x \cdot Cos\alpha - y \cdot Sin\alpha$$

$$y' = x \cdot Sin\alpha + y \cdot Cos\alpha$$

Одним из наиболее часто встречающихся способов ускорения операции поворота является отказ от вычисления синуса и косинуса угла во время выполнения программы, и использование их заранее подсчитанных значений, которые занесены в специальную таблицу. Например, в этой таблице могут храниться значения синусов и косинусов углов поворота с шагом в 1 градус. Тогда целое количество градусов угла поворота может служить в качестве индекса при извлечении соответствующих значений синусов

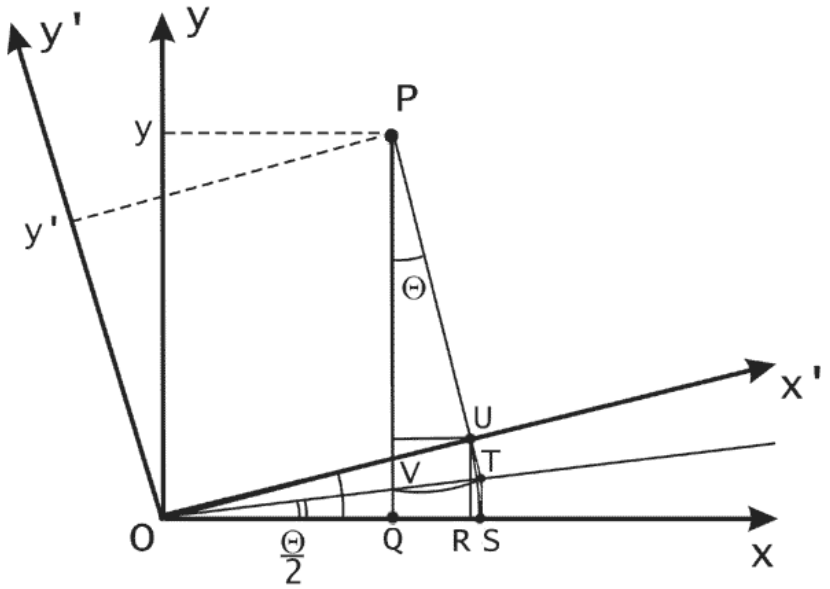

Рис. 14. Вывод формулы О. Бьюнемана.

и косинусов из таблицы. Такой прием называется табличным поворотом.

Дополнительным способом ускорения операции поворота является уменьшение количества операций умножения. Рассмотрим вывод формулы О. Бьюнемана с использованием тангенса половинного угла, в которой поворот точки вокруг начала координат производится за 3 операции умножения и 3 операции сложения. Так как на многих микропроцессорах операции умножения выполняются дольше чем операции сложения, то экономия времени достигается за счет уменьшения операций умножения.

Вывод формулы будем получать из геометрических построений, как показано на рис.28.

Будем искать выражение координат x и y через x' и y'. На оси Ox отложим отрезок OS, такой что $OS = x'$. Тогда $x = OS - QS = x' - QS$. Здесь отрезок QS является

горизонтальной проекцией отрезка PT, где $PT = PU + UT$, $PU = y'$, $UT = x' tg\frac{\theta}{2}$ $\Rightarrow x = x' - PT \cdot Sin\theta$, где $PT = y' + x' tg\frac{\theta}{2}$. Теперь, зная x, можно выразить y в виде суммы длин отрезков QV и VP. Так как длины отрезков PV и PT равны как радиусы окружности с центром в точке P, то $y = x \cdot tg\frac{\theta}{2} + PT$. Обозначим $PT = T^*$, отсюда следует, что

$$T^* = y' + x' tg\frac{\theta}{2},$$
$$x = x' - T^* Sin\theta,$$
$$y = x \cdot tg\frac{\theta}{2} + T^*$$

Последние три равенства будем называть формулой О. Бьюнемана.

Глава 4. Алгоритмы растровой графики

Рисование отрезков прямых

Растром называется прямоугольная сетка точек, формирующих изображение на экране компьютера. Каждая точка растра характеризуется двумя параметрами: своим положением на экране и своим цветом, если монитор цветной, или степенью яркости, если монитор черно-белый. Поскольку растровые изображения состоят из множества дискретных точек, то для работы с ними необходимы специальные алгоритмы. Рисование отрезка прямой линии - одна из простейших задач растровой графики. Смысл ее заключается в вычислении координат пикселов, находящихся вблизи непрерывных отрезков, лежащих на двумерной растровой сетке.

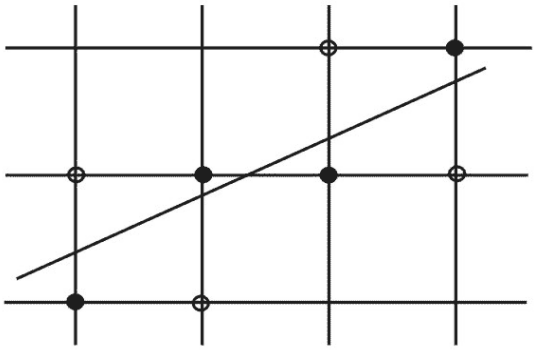

Рис. 15. Растеризация отрезка прямой линии.

Термин “пиксел” образован от английского pixel (picture element - элемент изображения) - то есть точка на экране. Будем считать, что пикселы имеют целочисленные координаты. На первый взгляд кажется, что эта задача имеет простое решение. Пусть конечные точки отрезка имеют целочисленные координаты, и уравнение прямой, содержащей отрезок: $y = kx + b$. Не нарушая общности, будем также считать, что тангенс угла наклона прямой лежит в пределах от 0 до 1. Тогда для изображения отрезка на растре достаточно для всех целых x, принадлежащих отрезку, выводить на экран точки с координатами $(x, \boldsymbol{Round}(y))$. Однако в этом методе присутствует операция умножения kx. Хотелось бы иметь алгоритм без частого использования операции умножения вещественных чисел. Избавиться от операции умножения можно следующим образом. Поскольку $k = \Delta y / \Delta x$, то один шаг по целочисленной сетке на оси x будет соответствовать $\Delta x = 1$. Отсюда получаем, что y будет увеличиваться на величину k. Итерационная последовательность выглядит следующим образом:

$$x_{i+1} = x_i + 1, \;\; y_{i+1} = y_i + k$$

Когда $k > 1$, то шаг по x будет приводить к шагу по $y > 1$, поэтому x и y следует поменять ролями, придавая y единичное приращение, а x будет увеличиваться на $\Delta x = \Delta y / k = 1 / k$ единиц. Этот алгоритм все же не свободен от операций с вещественными числами. Наиболее изящное решение задачи растровой развертки

отрезков прямых было найдено Брезенхемом. В его алгоритме вообще не используются операции с вещественными числами, в том числе операции умножения и деления.

Для вывода формул алгоритма Брезенхема рассмотрим рис. 30.

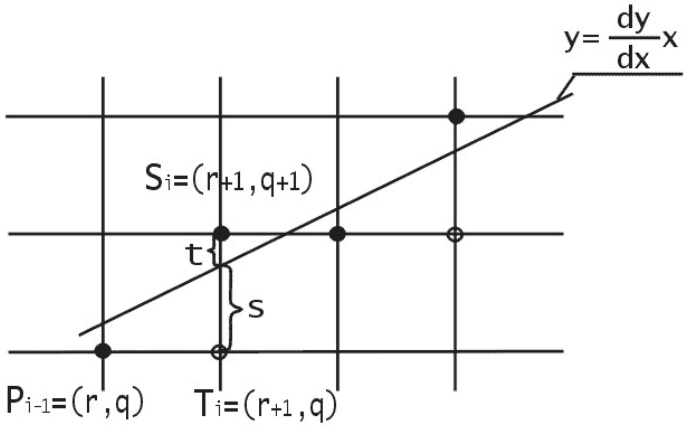

Рис. 30. Рисование отрезков прямых по методу Брезенхема.

Пусть начало отрезка имеет координаты $(x1, y1)$, а конец $(x2, y2)$. Обозначим $dx = (x2 - x1)$, $dy = (y2 - y1)$. Не нарушая общности, будем считать, что начало отрезка совпадает с началом координат, и прямая имеет вид $y = \frac{dx}{dy} x$, где $\frac{dx}{dy} \in [0,1]$.

Считаем что начальная точка находится слева. Пусть на $(i-1)$-м шаге текущей точкой отрезка является $P_{i-1} = (r, q)$. Выбор следующей точки S_i или T_i зависит от знака разности $(s - t)$. Если $(s - t) < 0$, то $P_i = T_i = (r + 1, q)$ и тогда $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$, если же $(s - t) \geq 0$, то $P_i = S_i = (r + 1, q + 1)$ и тогда $x_{i+1} = x_i + 1$, $y_{i+1} = y_i + 1$.

$$s = \frac{dy}{dx}(r+1) - q\,,\; t = q + 1 - \frac{dy}{dx}(r+1), \Rightarrow$$

$$s - t = 2\frac{dy}{dx}(r+1) - 2q - 1 \;\Rightarrow$$

$$dx(s-t) = 2(r \cdot dy - q \cdot dx) + 2dy - dx\,.$$

Поскольку знак $dx(s-t)$ совпадает со знаком разности $(s-t)$, то будем проверять знак выражения $d_i = dx(s-t)$. Так как $r = x_{i-1}$ и $q = y_{i-1}$, то $d_{i+1} = d_i + 2dy - 2dx(y_i - y_{i-1})$.

Пусть на предыдущем шаге $d_i < 0$, тогда $(y_i - y_{i-1}) = 0$ и $d_{i+1} = d_i + 2dy$. Если же на предыдущем шаге $d_i \geq 0$, то $(y_i - y_{i-1}) = 1$ и $d_{i+1} = d_i + 2(dy - dx)$.

Осталось узнать, как вычислить d_1. Так как при $i = 1$:

$$(x_0, y_0) = (0,0), \Rightarrow d_1 = 2dy - dx\,.$$

Далее приводится листинг процедуры на языке Паскаль, реализующей алгоритм Брезенхема.

```
Procedure Bresenham(x1,y1,x2,y2,Color: integer);
var
dx,dy,incr1,incr2,d,x,y,xend: integer;
begin
  dx:= ABS(x2-x1);
  dy:= Abs(y2-y1);
  d:=2*dy-dx;       {начальное значение для d}
  incr1:=2*dy;      {приращение для d<0}
  incr2:=2*(dy-dx); {приращение для d>=0}
   if x1>x2 then    {начинаем с точки с меньшим знач. x}
 begin
  x:=x2;
  y:=y2;
  xend:=x1;
end
else
begin
  x:=x1;
  y:=y1;
  xend:=x2;
end;
 PutPixel(x,y,Color);    {первая точка отрезка}
 While x<xend do
  begin
     x:=x+1;
   if d<0 then
     d:=d+incr1          {выбираем нижнюю точку}
   else
     begin
      y:=y+1;
      d:=d+incr2;   {выбираем верхнюю точку, y-возрастает}
end;
   PutPixel(x,y,Color);
  end;{while}
end;{procedure}
```

Отсечение

Перед тем, как исследовать методы получения изображений более сложных, чем отрезки прямых, рассмотрим проблему, незримо присутствующую в большинстве задач компьютерной графики. Эта проблема отсечения изображения по некоторой границе, например, по границе экрана, или, в общем случае, некоторого прямоугольного окна. Рассмотрим эту задачу применительно к отрезкам прямых. Некоторые из них полностью лежат внутри области экрана, другие целиком вне ее, а некоторые пересекают границу экрана. Правильное отображение отрезков означает нахождение точек пересечения их с границей экрана и рисование только тех их частей, которые попадают на экран. Один из очевидных способов отсечения отрезков состоит в определении точек пересечения прямой, содержащей отрезок, с каждой из четырех прямых, на которых лежат границы окна и проверки не лежит ли хотя бы одна точка пересечения на границе. В этом случае для каждой пары сторона-отрезок необходимо

решать систему из двух уравнений, используя операции умножения и деления. При этом удобно параметрическое задание прямых:

$$x = x1 + t(x2 - x1)$$
$$y = y1 + t(y2 - y1).$$

Для $t \in [0,1]$ эти уравнения определяют точки, находящиеся между $(x1, y1)$ и $(x2, y2)$. Специальной проверки требует случай, когда отрезок параллелен стороне окна. Пусть координата *x* точки пересечения найдена, тогда

$$t = \frac{x - x_1}{x_2 - x_1} \Rightarrow y = y_1 + \frac{x - x_1}{x_2 - x_1}(y_2 - y_1)$$

Рассмотрим алгоритм Коэна-Сазерленда для отсечения отрезков прямых. Этот алгоритм позволяет легко определять нахождение отрезка полностью внутри или полностью снаружи окна, и если так, то его можно рисовать или не рисовать, не заботясь об отсечении по границе окна.

Для работы алгоритма вся плоскость в которой лежит окно разбивается на девять подобластей или квадрантов, как показано на рис. 31.

Окну соответствует область обозначенная кодом 0000. Конечным точкам отрезка приписывается 4-битный код “вне/внутри” в зависимости от нахождения отрезка в соответствующей подобласти. Каждому биту присваивается значение 1 в соответствии со следующим правилом.

Бит 1 - точка находится выше окна;

Бит 2 – точка находится ниже окна;

Бит 3 - точка находится справа от окна;

Бит 4 - точка находится слева от окна;

Иначе биту присваивается нулевое значение. Значения этих битов для конечных точек отрезков легко определить по знакам соответствующих разностей: $(y_{\max} - y)$ - для 1-го бита, $(y - y_{\min})$ - для 2-го бита, $(x_{\max} - x)$ - для 3-го бита и $(x - x_{\min})$ - для 4-го бита. Отрезок рисуется без отсечения, то есть принимается целиком, если оба кода равны 0000, или $[кодP1] \text{ИЛИ} [кодP2] = 0000$, где ИЛИ – бинарная операция. Отрезок отбрасывается без вычислений если оба его конца находятся выше, ниже, правее или левее окна. В этих случаях соответствующие биты в обоих кодах равны 1 и это легко определить, умножив эти коды по бинарной операции И. Если результат операции И равен 0000, то отрезок нельзя ни принять ни отбросить, так как он может пересекаться с окном. В этом случае применяется последовательное разделение отрезка, так что на каждом шаге конечная точка отрезка с ненулевым кодом

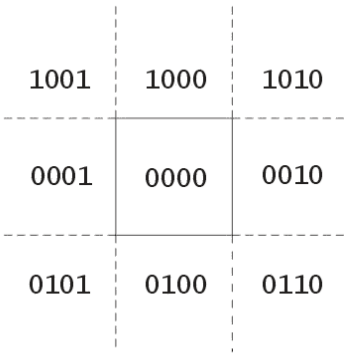

Рис. 31. Разбиение на подобласти в методе Коэна-Сазерленда.

вне/внутри заменяется на точку, лежащую на стороне окна или на прямой содержащей сторону. При этом порядок перебора сторон окна не имеет значения.

Далее приводится текст процедуры на языке Паскаль, с довольно изящной реализацией этого метода. Отрезок задан граничными точками $P1=(x1,y1)$, $P2=(x2,y2)$, границы окна: xmin, xmax, ymin, ymax. Используются вызовы процедур: Accept_Check – выполняет проверку на полное принятие отрезка; Reject_Check – на полный отказ от рисования отрезка; Outcodes – вычисляет 4-х битовый код “вне/внутри”; SWAP – меняет местами координаты двух точек.

```
Procedure CLIP(x1,x2,y1,y2,xmin,xmax,ymin,ymax: real);
type
 outcode = array[1..4] of boolean;
var
 accept,reject,done: boolean;
outcode1,outcode2,
outcode3,outcode4:outcode;{коды вне/внутри}
begin
 accept:= false;
 reject:= false;
 done:= false;
repeat
Outcodes(x1,y1,outcode1);
Outcodes(x2,y2,outcode2);
{проверка на отбрасывание}
 reject:=Reject_Check(outcode1,outcode2);
  if reject then done:= true
  else
     begin {возможно принятие целиком}
          accept:=Accept_Check(outcode1,outcode2);
     if accept then done:=true
     else
begin {разделить отрезок}
{если P1 внутри, то с помощью SWAP сделать снаружи}
if not((outcode1[1])or(outcode1[2])or
       (outcode1[3])or(outcode1[4])) then SWAP;
{теперь P1 перемещается в точку пересечения}
  if outcode1[1] then
   begin {отбросить верхнюю часть}
     x1:=x1+(x2-x1)*(ymax-y1)/(y2-y1);
     y1:=ymax;
   end
else if outcode1[2] then
if outcode1[1] then
   begin {отбросить нижнюю часть}
     x1:=x1+(x2-x1)*(ymin-y1)/(y2-y1);
     y1:=ymin;
   end
else if outcode1[3] then
   begin {отбросить правую часть}
     y1:=x1+(y2-y1)*(ymax-x1)/(x2-x1);
     x1:=xmax;
```

```
   end
else if outcode1[4] then
   begin {отбросить левую часть}
     y1:=x1+(y2-y1)*(ymin-x1)/(x2-x1);
     x1:=xmin;
   end;
  end;
 end;
until done;
  if accept then
   Line(x1,y1,x2,y2); {нарисовать отрезок}
end;{procedure}
```

Глава 5. Нормирующие преобразования видимого объема

Видимый объем

Зададим центральную перспективную проекцию с центром проекции в начале координат, как показано на рис. 32. Для реальных вычислений необходимо также определить значения минимальной и максимальной отсекающих плоскостей по

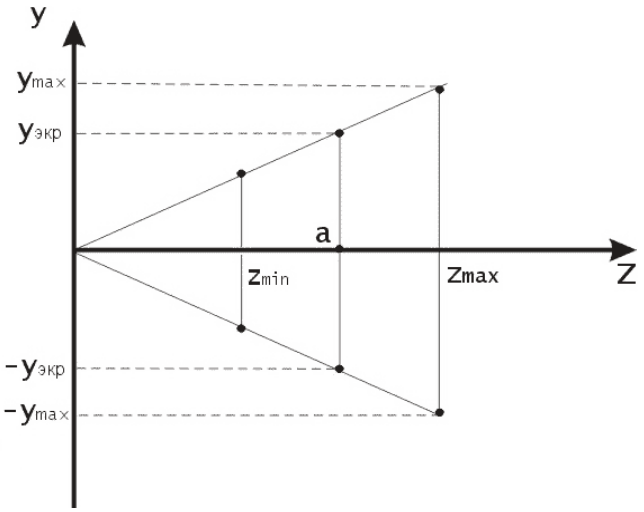

Рис. 32. Видимый объем, вид сбоку.

координате z: $z = z_{\min}$ и $z = z_{\max}$, соответственно.

Границы экрана, или окна вывода задают четыре отсекающих плоскости сверху, снизу, справа и слева. Таким образом, изображение, получаемое с помощью нашей проекции может находится только внутри усеченной пирамиды образованной упомянутыми плоскостями, причем объекты вне этой пирамиды не проецируются на экран, т.е. являются невидимыми для наблюдателя. Видимым объемом называется замкнутая область пространства, объекты внутри которой проецируются на экран. В случае центральной перспективной проекции видимым объемом является усеченная пирамида.

Одной из важных задач компьютерной графики является нахождение эффективного способа отсечения трехмерных объектов по границе видимого объема и удаление невидимых ребер и граней. Например, в случае центральной перспективы, для решения задачи отсечения пришлось бы для каждой грани или ребра находить точки пересечения с плоскостями усеченной пирамиды, что в общем случае потребовало бы значительных вычислений. Решение заключается в преобразовании видимого объема к виду, в котором вычисления проводились бы значительно проще. В общем идея заключается в том, чтобы свести преобразование центральной перспективы математически к виду параллельной проекции, в которой, очевидно, операция взятия проекции сводится к простому отбрасыванию у точек координаты z.

Нормирование

Будем решать задачу в два этапа. В начале приведем видимый объем к нормированному виду. При этом значение $z_{\max} = 1$, а границы по осям x и y лежат в диапазоне $[-1,1]$, как показано на рис. 33.

Нормирующим преобразованием в этом случае будет операция масштабирования, которая для произвольной точки X выражается в виде:

$$X' = X \cdot S\left(1/x_{\max}, 1/y_{\max}, 1/z_{\max} \right),$$

где $\frac{x_{\max} a}{z_{\max}} = x_{экр} \Rightarrow x_{экр} = \frac{z_{\max} x_{экр}}{a}$, и соответственно, $y_{экр} = \frac{z_{\max} y_{экр}}{a}$.

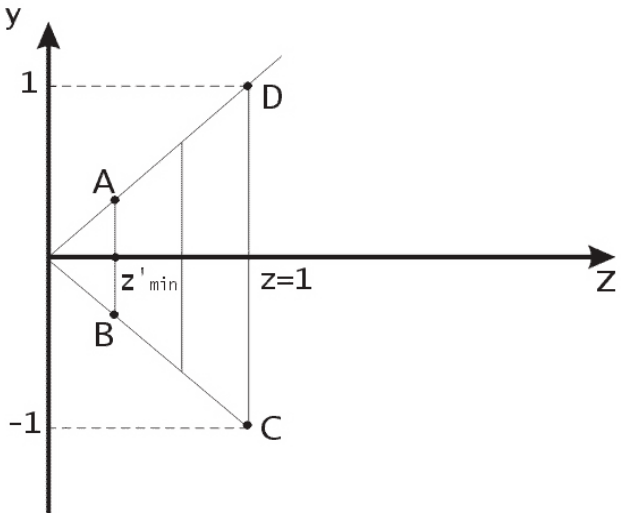

Рис. 16. Нормированный видимый объем.

Нормированный видимый объем позволяет с большей легкостью решать задачу отсечения по границе. А именно, в этом случае может применяться модифицированный вариант алгоритма Коэна-Сазарленда в котором вместо 4-битовых используются 6-битовые коды вне/внутри для описания нахождения точки в соответствующей области пространства. Уравнения боковых граней видимого объема сильно упрощаются, например, для правой отсекающей плоскости уравнение запишется $z = x$, а для левой боковой $z = -x$ и т.д. . Тогда для некоторой точки (x, y, z) условие установления бита в единицу будет следующим:

1-й бит: $y > z$

2-й бит: $y < -z$

3-й бит: $x > z$

4-й бит: $x < -z$

5-й бит: $z > 1$

6-й бит: $z < z'_{\min}$

Для эффективного решения задачи удаления невидимых ребер/граней преобразуем нормированный видимый объем к каноническому виду, как показано на рис. 34.

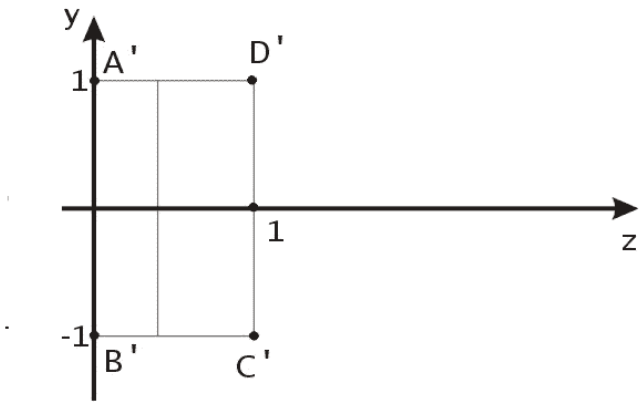

Рис. 17. Канонический видимый объем.

Это достигается с помощью матрицы

$$M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1-z'_{\min}} & 1 \\ 0 & 0 & \dfrac{-z'_{\min}}{1-z'_{\min}} & 0 \end{bmatrix}.$$

После применения матрицы M_p нормированный видимый объем становится прямоугольным параллелепипедом, что позволяет перейти от центральной перспективной к параллельной проекции. Легко проверить, что как показано на рис. 33 и рис. 34: $D' = DM_p$, $A' = AM_p$, $B' = BM_p$, $C' = CM_p$, а также, например, $[z'_{\min}, z'_{\min}, z'_{\min}, 1] \xrightarrow{M_p} [1,1,0,1]$.

Итак, нормирующие преобразования видимого объема могут производиться за два шага.

1 шаг - преобразование к нормированному видимому объему и отсечение по 3-х мерному алгоритму Коэна-Сазерленда.

2 шаг - преобразование к прямоугольному параллелепипеду с помощью матрицы M_p и удаление скрытых поверхностей при условии равенства координат x и y.

Глава 6. Алгоритмы удаления невидимых ребер и граней

Классификация

Алгоритмы удаления невидимых граней могут быть условно поделены на два класса в зависимости от принципов, заложенных для их реализации. Первый класс – это алгоритмы работающие в пространстве объекта. Это означает, что для определения видимости данной грани сравнивается ее взаимное расположение со всеми остальными гранями в трехмерной сцене. Пусть N – количество граней в трехмерной сцене. Для построения трехмерной сцены в этом случае необходимо сравнить положение каждой грани с оставшимися, что требует порядка N^2 операций. Например, пусть количество граней в трехмерной сцене $N = 1000$, тогда время работы алгоритмов этого класса порядка 1,000,000 операций.

Другой класс алгоритмов - работающих в пространстве изображения, основан на нахождении точки ближайшей грани, которую пересекает луч зрения, проходящий через заданную точку на растре. Поскольку число точек на растровом экране фиксировано, то алгоритмы этого класса менее чувствительны к увеличению количества объектов в трехмерной сцене. Пусть n - число точек на растровом экране. Тогда количество операций, необходимых для построения трехмерной сцены будет порядка $n \cdot N$. Например, для экранного разрешения 320×200 точек, $n = 64000$, тогда количество операций для $N = 1000$ граней будет порядка 64,000,000. Выбор класса алгоритма может зависеть от особенностей конкретной задачи, а также от способов реализации алгоритма.

Алгоритм с использованием z-буфера

Рассмотрим алгоритм удаления невидимых граней с использованием z-буфера, который является одним из наиболее часто используемых в современных приложениях компьютерной графики. Он работает в пространстве изображения и применяется в таких популярных графических библиотеках как OpenGL и Direct3D.

Алгоритм работает в параллельной проекции. Пусть размеры окна вывода или экрана составляют X точек в ширину и Y точек в высоту. В качестве z-буфера заведем двумерный прямоугольный массив чисел по размерности совпадающий с окном вывода или экрана, т.е. $X \times Y$. В z-буфере будут храниться текущие значения z-координат каждого пиксела.

В начале работы алгоритма в z-буфер заносятся значения, соответствующие бесконечности. Каждая грань трехмерного объекта, представленная в виде многоугольника, преобразуется в растровую форму. При разложении в растр для каждой точки многоугольника вычисляется значение ее z-координаты. Если z-координата оказалась меньше чем текущее значение в z-буфере, то в z-буфер заносится z-координата точки, и на экране рисуется точка цветом текущего многоугольника. После разложения в растр всех многоугольников изображение трехмерной сцены построено.

Рассмотрим способ ускоренного вычисления z-координат при разложении многоугольников в растр. Запишем уравнение плоскости, образуемой многоугольником в пространстве: $Ax + By + Cz + D = 0$.

Выразим z-координату точки: $z = \frac{-D - Ax - By}{C} = f(x, y)$. Пусть $z_0 = f(x_0, y_0)$. Найдем z-координату для соседней точки $z_1 = f(x_0 + \Delta x, y_0) = \frac{-D - A(x_0 + \Delta x) - By_0}{C} = \frac{-D - Ax_0 - By_0}{C} - \frac{A\Delta x}{C} =$ $= z_0 - \frac{A\Delta x}{C}$. Для соседнего пиксела на экране $\Delta x = 1$, тогда $\frac{A\Delta x}{C} = Const$, отсюда следует, что $z_1 = z_0 - Const$. Таким образом, вычисление z-координаты соседнего пиксела сводится к одной операции вычитания.

Метод сортировки по глубине

Рассмотрим далее алгоритм удаления невидимых граней методом сортировки по глубине (авторы: Ньюэлл, Ньюэлл, Санча). Часть этого метода работает в пространстве объекта, а часть в пространстве изображения. Он также работает для параллельной проекции, то есть с учетом того что произведено перспективное преобразование. Введем определение пространственной оболочки.

Пространственной оболочкой трехмерного объекта называется минимальный прямоугольный параллелепипед, целиком содержащий внутри себя данный объект. Аналогично можно определить двумерную и одномерную пространственные оболочки.

Метод состоит из трех основных шагов:

1. Упорядочение всех многоугольников в соответствии с их наибольшими z-координатами.
2. Разрешение всех неопределенностей, которые возникают при перекрытии z-оболочек многоугольников.
3. Преобразование каждого из многоугольников в растровую форму, производимое в порядке уменьшения их наибольшей z-координаты.

Ближайшие многоугольники преобразуются в растровую форму последними и закрывают более отдаленные многоугольники, так как изображаются поверх предыдущих. Реализация пунктов 1 и 3 достаточно очевидна. Рассмотрим подробнее пункт 2.

Пусть многоугольник P после упорядочения находится в конце списка, то есть является наиболее удаленным. Все многоугольники Q чьи оболочки перекрываются с z-оболочкой P должны проходить проверку по пяти тестам (шагам). Если на некотором шаге получен утвердительный ответ, то P сразу преобразуется в растровую форму.

Пять тестов:

1. x-Оболочки многоугольников не перекрываются, поэтому сами многоугольники тоже не перекрываются.
2. y-Оболочки многоугольников не перекрываются, поэтому сами многоугольники тоже не перекрываются.
3. P полностью расположен с той стороны от плоскости Q, которая дальше от точки зрения (этот тест дает положительный ответ как показано на рис. 36.
4. Q полностью расположен с той стороны от плоскости P, которая ближе к точке зрения. Этот тест дает положительный ответ как показано на рис. 37.

5. Проекции многоугольников на плоскости xOy, то есть на экране, не перекрываются (это определяется сравнением ребер одного многоугольника с ребрами другого).

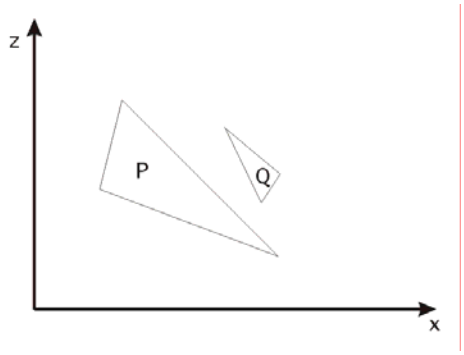

Рис. 18. *Z* -оболочки треугольников P и Q – пересекаются.

Если во всех пяти тестах получен отрицательный ответ, то P – действительно закрывает Q. Тогда меняем P и Q в списке местами. В случае, как показано на рис. 38, алгоритм зацикливается.

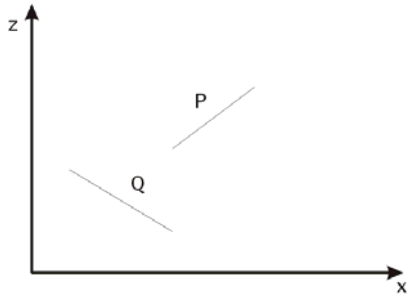

Рис. 20

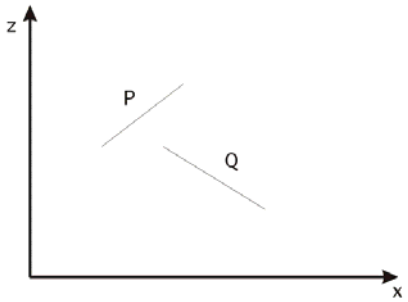

Рис. 19

Для избежания зацикливания вводится ограничение: многоугольник, перемещенный в конец списка (т.е. помеченный), не может быть повторно перемещен. Вместо этого

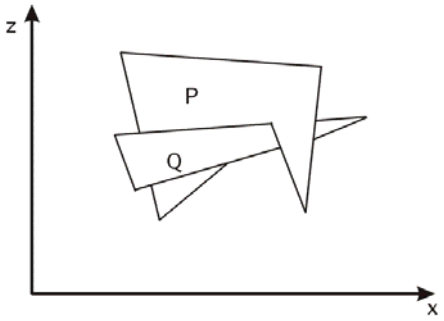

Рис. 21

многоугольник P или Q разделяется плоскостью другого на два новых многоугольника. Эти два новых многоугольника включаются в соответствующие места упорядоченного списка, и алгоритм продолжает работу.

Метод удаления невидимых граней выпуклых тел

В отличие от универсальных алгоритмов узкоспециализированный алгоритм удаления невидимых граней выпуклых тел позволяет производить вычисления гораздо быстрее. Он работает для центральной перспективной проекции. Рассмотрим работу этого

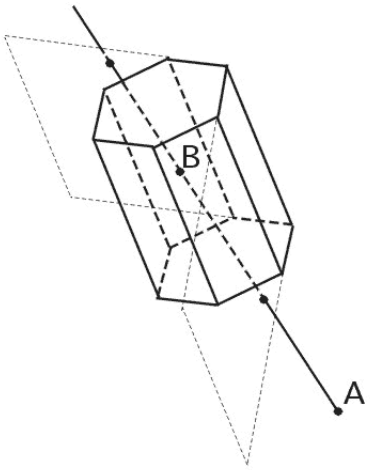

Рис. 22. Пересечения прямой AB с плоскостями граней призмы.

алгоритма на примере как изображено на рис. 39.

Пусть наблюдатель находится в точке *A*. Выберем точку *B*, которая заведомо является внутренней для выпуклой фигуры, в данном случае призмы. Выберем некоторую грань, про которую мы хотим узнать видима она из точки *A*, или не видима. Построим плоскость, в которой лежит выбранная грань. Найдем точку пересечения плоскости и прямой, которая образована отрезком *AB*. Если точка пересечения прямой и плоскости лежит внутри отрезка *AB*, то делаем вывод, что данная грань видима. Если точка пересечения находится вне отрезка *AB*, то грань не видима. В случае, когда прямая и плоскость параллельны, считаем что грань не видима.

Глава 7. Модели расчета освещенности граней трехмерных объектов

Цветовой куб RGB

Основной характеристикой света в компьютерной графике является яркость. Поскольку яркость является субъективным понятием, основанным на человеческом восприятии света, то для численных расчетов применяется термин интенсивность, что соответствует яркости и является энергетической характеристикой световой волны. В расчетах интенсивность обычно принимает значения от 0 до 1. При этом интенсивность равна нулю при полном отсутствии света, а значение 1 соответствует максимальной яркости.

В компьютерной графике для расчета освещенности граней объектов зачастую применяется трехкомпонентная цветовая модель “Красный, Зеленый, Синий”, что в английском варианте записывается RGB (Red, Green, Blue). Эта модель позволяет задавать любой цвет в виде трех компонент интенсивностей базовых цветов: красного, зеленого и синего. Интенсивность отраженного света точек пространственных объектов вычисляют отдельно для каждой их трех составляющих цветовых компонент, а затем объединяют в результирующую тройку цветов. Далее будем считать, что примеры расчета интенсивностей отраженного света применяются к каждому их трех базовых цветов.

Наглядное представление цветовой модели RGB возможно с помощью так называемого цветового куба. По осям трехмерной декартовой системы координат откладываются значения интенсивностей компонент красной, зеленой и синей составляющих в диапазоне от 0 до 1. Весь диапазон цветов представляется как множество векторов в пределах единичного куба в пространстве (R,G,B). Главная диагональ этого куба состоит из векторов, которые соответствуют всевозможным градациям серого цвета от черного до белого. Черному цвету соответствует вектор (0,0,0), белому цвету соответствует вектор (1,1,1).

Вершины цветового куба RGB представляют собой цвета максимальной чистоты. Все они имеют свои названия. Первые три, красный, зеленый и синий, называются основными цветами. Остальные три вершины куба, за исключением черного и белого цветов, получаются за счет сложения пар основных цветов. Эти цвета формируют тройку дополнительных цветов CMY или Cyan-Magenta-Yellow, Бирюзовый или цвет морской волны, лиловый и желтый. Система цветов RGB называется аддитивной, за счет того что произвольный цвет в ней формируется как сумма трех основных компонент. Цветовая модель CMY также называется субтрактивной, (от английского Subtraction – вычитание), за счет того что цветовая точка получается в ней вычитанием значений интенсивностей цветовых компонент из белого цвета, то есть из вектора (1,1,1) в системе RGB.

Субтрактивная модель находит широкое применение в полиграфии, только там она называется CMYK и состоит из четырех цветов. Буква K в этой аббревиатуре соответствует черному цвету. По-английски черный пишется как “Black”. Поскольку буква B уже используется для обозначения синего цвета, то для обозначения черного используется последняя буква в английском варианте написания слова. Черный цвет добавляется из соображений удобства применения технологии. Лист бумаги обычно имеет белый цвет, именно поэтому субтрактивная модель CMYK нашла столь широкое применение в книгопечатании.

По координатам (r,g,b) вектора из цветового куба можно определить другую тройку цветовых параметров из пространства тон-яркость-насыщенность или HVS (Hue–Value–Saturation, англ.). Яркость цвета вычисляется как расстояние от начала координат до точки пересечения перпендикуляра, опущенного из точки с координатами (r,g,b) на главную диагональ цветового куба. Насыщенность цвета характеризует его чистоту, удаленность от ближайшей градации серого цвета. Таким образом, насыщенность цвета соответствует длине перпендикуляра, опущенного из точки (r,g,b) на главную диагональ цветового куба. Цветовой тон в цветовом кубе определяется в соответствии с близостью к одному из основных или дополнительных цветов. Цветовой тон соответствует обще используемому понятию цвета. Например, при сравнении цвета листьев различных пород деревьев, мы говорим, что все они “зеленые”, хотя некоторые из них могут иметь более темную или светлую окраску. То есть цвета листьев могут и не совпадать в точности, а лишь иметь общий цветовой тон, хотя мы им приписываем один и тот же цвет.

Эмпирическая модель расчета освещенности

При расчете освещенности граней применяют следующие типы освещения и отражения света от поверхностей.

- Рассеянное
- Диффузное
- Зеркальное

Интенсивность освещения граней трехмерных объектов рассеянным светом считается постоянной в любой точке пространства. Она обусловлена множественными отражениями света от всех объектов в пространстве. При освещении трехмерного

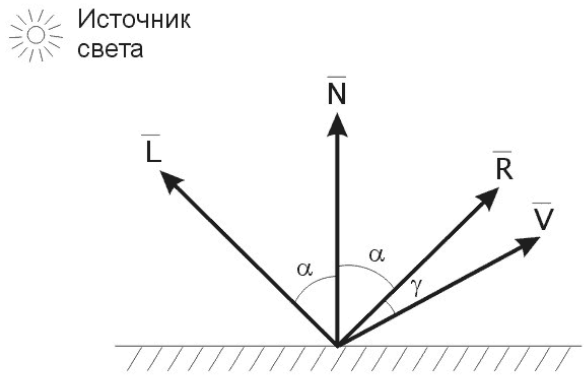

Рис. 40. Расчет интенсивности отраженного света.

объекта рассеянным светом интенсивность отраженного света вычисляется как $I_a = I_p \cdot k_a$, где I_p - интенсивность падающего света, $k_a \in [0,1]$ - коэффициент рассеянного отражения, зависит от отражающих свойств материала грани.

Для расчета интенсивности диффузного отражения света может применяться закон косинусов Ламберта: $I_d = I_p \cdot k_d \cdot Cos(\alpha)$, где α - угол падения, рассчитывается как угол между направлением на источник света и нормалью к поверхности. Пусть направление на источник света представлено единичным вектором $\overline{L}$, а $\overline{N}$ - единичный вектор

нормали. Тогда $Cos\alpha = \left(\overline{L},\overline{N}\right)$ - скалярное произведение векторов. Тогда $I_d = I_p \cdot k_d \cdot \left(\overline{L}, \overline{N}\right)$, где k_d - коэффициент диффузного отражения.

Вычисление зеркально отраженного света производится также с помощью различных эмпирических моделей, которые позволяют учитывать реальную шероховатость поверхностей. Например, в модели, предложенной Фонгом, интенсивность зеркально отраженного света рассчитывается в зависимости от степени отклонения от истинного значения вектора зеркально отраженного луча света. Пусть $\overline{R}$ - вектор зеркально отраженного луча света, а $\overline{V}$ - вектор, определяющий направление на наблюдателя. Тогда интенсивность зеркально отраженного света по модели Фонга рассчитывается так: $I_m = I_p \cdot k_m Cos^n\gamma$, где γ - угол между векторами $\overline{R}$ и $\overline{V}$. Константа *n* – может принимать значения от 1 до примерно 200, в зависимости от отражающей способности материала. Большим значениям *n* соответствует большая степень “гладкости” или “зеркальности” поверхности. Если векторы $\overline{R}$ и $\overline{V}$ - нормированы, то формула преобразуется к виду: $I_m = I_p \cdot k_m\left(\overline{R},\overline{V}\right)^n$.

Интенсивность отраженного света уменьшается обратно пропорционально квадрату расстояния от источника до наблюдателя. Поэтому можно записать формулу расчета интенсивности отраженного луча света для трех составляющих: рассеянного, диффузного и зеркального отражения с учетом расстояния:

$$I = I_p k_a + \frac{I_p}{R + r^2}\left(k_d \cdot \left(\overline{L},\overline{N}\right) + k_m\left(\overline{R},\overline{V}\right)^n\right),$$

где r - расстояние от точки отражения до наблюдателя, а $R \geq 1$ - некоторая константа. Иногда, для ускорения вычислений, берут не вторую, а первую степень расстояния r.

В системах компьютерной визуализации также учитываются такие свойства материалов отражающих поверхностей как прозрачность, преломление и свечение. Степень прозрачности материала грани может описываться с помощью константы, принимающей значение от нуля до единицы, причем значение 1 соответствует полной непрозрачности материала грани. Пусть интенсивности отраженного света двух перекрывающихся поверхностей равны I_1 и I_2. Пусть первая поверхность находится ближе к наблюдателю и является полупрозрачной с коэффициентом прозрачности α. Тогда суммарная интенсивность отраженного света может быть вычислена как взвешенное среднее: $I = I_1\alpha + I_2(1-\alpha)$.

Модели для вычисления эффектов преломления и свечения здесь не рассматриваются.

Глава 8. Кубические сплайны

Сплайновая функция

Рассмотрим задачу проведения гладких кривых по заданным граничным точкам, или задачу интерполяции. Поскольку через две точки можно провести сколь угодно много гладких кривых, то для решения этой задачи необходимо ограничить класс функций, которые будут определять искомую кривую. Математическими сплайнами называют функции, используемые для аппроксимации кривых. Важным их свойством является простота вычислений. На практике часто используют сплайны вида полиномов третьей степени. С их помощью довольно удобно проводить кривые, которые интуитивно

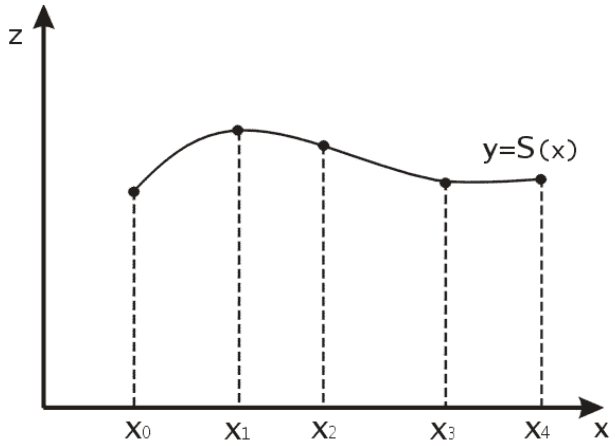

Рис. 41. Сплайновая функция.

соответствуют человеческому субъективному понятию гладкости. Термин “сплайн” происходит от английского spline – что означает гибкую полоску стали, которую применяли чертежники для проведения плавных кривых, например, для построения обводов кораблей или самолетов.

Рассмотрим в начале сплайновую функцию для построения графика функции одной переменной. Пусть на плоскости задана последовательность точек $\{x_i, y_i\}, i = \overline{0,m}$, причем $x_0 < x_1 <,..., x_{m-1} < x_m$. Определим искомую функцию $y = S(x)$, причем поставим два условия:

1) Функция должна проходить через все заданные точки: $S(x_i) = y_i$, $i = \overline{0,m}$.
2) Функция должна быть дважды непрерывно дифференцируема, то есть иметь непрерывную вторую производную на всем отрезке $[x_0, x_m]$.

На каждом из отрезков $[x_i, x_{i+1}]$, $i = \overline{0,m-1}$ будем искать функцию в виде полинома третьей степени:

$$S_i(x) = \sum_{j=0}^{3} a_{ij}(x - x_i)^j .$$

Задача построения полинома сводится к нахождению коэффициентов a_{ij}. Поскольку для каждого из отрезков $[x_i, x_{i+1}]$ необходимо найти 4 коэффициента a_{ij}, то всего количество искомых коэффициентов будет $4m$. Для нахождения всех коэффициентов определим соответствующее количество уравнений. Первые $(m-1)$ уравнений получаем из условий совпадения значений функции во внутренних узлах $x_i, i = \overline{1, m-1}$. Следующие $2(m-1)$ уравнений получаем аналогично из условий совпадения значений первых и вторых производных во внутренних узлах. Вместе с первым условием получаем $m-1+m-1+m-1+m+1 = 4m-2$ уравнений. Недостающие два уравнения можно получить заданием значений первых производных в концевых точках отрезка $[x_0, x_m]$. Так могут быть заданы граничные условия.

Сплайновые кривые Эрмита и Безье

Перейдем к более сложному случаю – заданию кривых в трехмерном пространстве. В случае функционального задания кривой $\begin{cases} y = f(x) \\ z = f(x) \end{cases}$ возможны многозначности в случае самопересечений и неудобства при значениях производных равных ∞. Ввиду этого будем искать функцию в параметрическом виде. Пусть t - независимый параметр, такой что $0 \le t \le 1$. Кубическим параметрическим сплайном назовем следующую систему уравнений:

$$\begin{cases} x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \\ z(t) = a_z t^3 + b_z t^2 + c_z t + d_z \end{cases}$$

Координаты точек на кривой описываются вектором $(x(t), y(t), z(t))$, а три производные задают координаты соответствующего касательного вектора в точке. Например, для координаты x:

$$\frac{dx}{dt} = 3a_x t^2 + 2b_x t + c_x .$$

Одним из способов задания параметрического кубического сплайна является указание координат начальной и конечной точек, а также векторов касательных в них. Такой способ задания называется формой Эрмита. Обозначим концевые точки P_1 и P_4, а касательные векторы в них R_1 и R_4. Индексы выбраны таким образом с учетом дальнейшего изложения.

Будем решать задачу нахождения четверки коэффициентов a_x, b_x, c_x, d_x, так как для оставшихся двух уравнений коэффициенты находятся аналогично. Запишем условие для построения сплайна:

$$x(0) = P_{1x},\ x(1) = P_{4x},\ x'(0) = R_{1x},\ x'(1) = R_{4x} \qquad (*)$$

Перепишем выражение для x в векторном виде [3]:

$$x(t) = \left[t^3, t^2, t, 1\right] \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}_x .$$

Обозначим вектор строку $T = \left[t^3, t^2, t, 1\right]$ и вектор столбец коэффициентов $C_x = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}_x$,

тогда $x(t) = TC_x$.

Из (*) следует, что $x(0) = P_{1x} = [0,0,0,1]C_x$, $x(1) = P_{4x} = [1,1,1,1]C_x$. Для касательных $x'(t) = \left[3t^2, 2t, 1, 0\right]C_x, \Rightarrow$

$x'(0) = R_{1x} = [0,0,1,0]C_x$,

$x'(1) = R_{4x} = [3,2,1,0]C_x$. Отсюда получаем векторно-матричное уравнение:

$$\begin{bmatrix} P_{1x} \\ P_{4x} \\ R_{1x} \\ R_{4x} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} C_x .$$

Эта система решается относительно C_x нахождением обратной матрицы размером 4×4.

$$C_x = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{1x} \\ P_{4x} \\ R_{1x} \\ R_{4x} \end{bmatrix} = M_h G_{hx} .$$

Здесь M_h - эрмитова матрица, G_h - геометрический вектор Эрмита. Подставим выражение C_x для нахождения $x(t)$: $x(t) = TM_h G_{hx}$. Аналогично для остальных координат: $y(t) = TM_h G_{hy}$, $z(t) = TM_h G_{hz}$.

Выпишем в явном виде формулы для вычисления координат точек сплайна. Так как $TM_h = \left[\left(2t^3 - 3t^2 + 1\right), \left(-2t^3 + 3t^2\right), \left(t^3 - 2t^2 + t\right), \left(t^3 - t^2\right)\right]$, то умножая справа на G_{hx}, получаем:

$$x(t) = TM_h G_{hx} = P_{1x}\left(2t^3 - 3t^2 + 1\right) + P_{4x}\left(-2t^3 + 3t^2\right) + R_{1x}\left(t^3 - 2t^2 + t\right) + R_{4x}\left(t^3 - t^2\right).$$

Четыре функции в скобках называются функциями сопряжения.

Форму кривой, заданной в форме Эрмита, легко изменять если учитывать, что направление вектора касательной задает начальное направление, а модуль вектора касательной задает степень вытянутости кривой в направлении этого вектора, как показано на рис. 42.

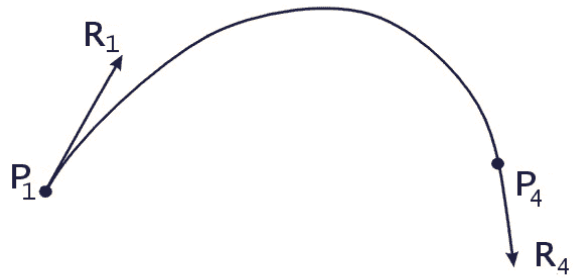

Рис. 42. Параметрический сплайн в форме Эрмита. Вытянутость кривой вправо обеспечивается тем, что $|R_1| > |R_4|$.

Рассмотрим форму Безье, которая отличается от формы Эрмита способом задания граничных условий, а именно, вместо векторов R_1 и R_4 вводятся точки (и соответствующие им радиус векторы) P_2 и P_3, как показано на рис.43, такие что выполняются условия: $P'(0) = R_1 = 3(P_2 - P_1)$ и $P'(1) = R_4 = 3(P_4 - P_3)$.

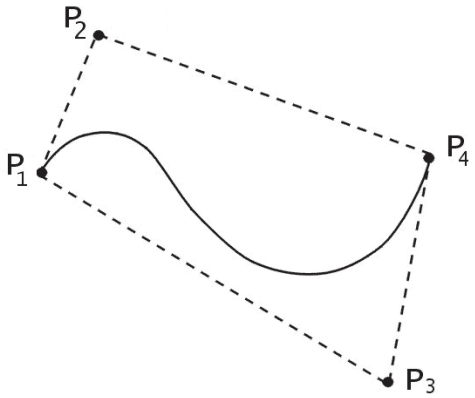

Рис. 23. Параметрический сплайн в форме Безье.

Переход от формы Эрмита к форме Безье осуществляется преобразованием:

$$G_h = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = M_{hb} G_b \ , \qquad (*)$$

где G_b - геометрический вектор Безье. Подставляя это в выражение для $x(t)$, получаем

$$x(t) = TM_h G_{hx} = TM_h M_{hb} G_{bx} = (1-t^3)P_1 + 3t(t-1)^2 P_2 + 3t^2(1-t)P_3 + t^3 P_4 \ .$$

Полезным свойством сплайнов в форме Безье является то что кривая всегда лежит внутри выпуклой оболочки, образованной четырехугольником $(P_1P_2P_3P_4)$. Это свойство можно доказать, пользуясь тем, что в выражении (*) коэффициенты принимают значения от 0 до 1 и их сумма равна единице.

Заметим, что матрица вида

$$M_h M_{hb} = M_b = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$ - называется матрицей Безье.

ЧАСТЬ 2. ПРИЛОЖЕНИЯ КОМПЬЮТЕРНОЙ ГРАФИКИ

Глава 10. Оконный интерфейс Windows

В этой главе вы получите представление о терминологии и общих принципах функционирования графического оконного интерфейса операционной системы Windows.

После изучения материалов по данной теме вы научитесь:

- Подключать и использовать программные средства уровня операционной системы в Delphi.
- Создавать и отображать на экране окна Windows.
- Пользоваться функциями построения графических изображений средствами Windows.

Система программирования Delphi позволяет создавать программы, пользуясь функциями уровня операционной системы – Windows API. Аббревиатура API расшифровывается как Application Programming Interface – Интерфейс Прикладного Программирования. Функции Windows API полностью обеспечивают программиста набором средств для разработки любых программ. В этот набор, в том числе, входят и функции для работы с окнами Windows. Важно понимать, что функции Windows API могут вызываться из любого языка программирования высокого уровня, для которого существует компилятор для Windows.

Замечание. Функции Windows API работают по принципу структурного программирования. В наборе функций Windows API не применяются объекты как в технологии объектно-ориентированного программирования.

Для чего использовать функции Windows API?

Необходимость применения функций Windows API может быть по следующим причинам. Современные системы программирования обычно применяют технологии быстрой разработки программ (Rapid Application Development, RAD). По этой технологии программисту предоставляется набор объектов и алгоритмов, которые выполняют большую часть рутинных действий незаметно для программиста. За счет этого программист получает возможность сконцентрировать усилия на разработке алгоритма решения прикладной задачи вместо решения часто повторяющихся задач общесистемного плана вроде установления размеров, вида и расположения окон на экране и элементов управления на них. Однако, технология RAD, за счет своей универсальности и гибкости, требует использования дополнительного программного кода, что может сказаться на быстродействии программы. Это особенно важно для приложений компьютерной графики, где вывод на экран может занимать большую часть времени работы программы. Кроме этого, практика в применении функций Windows API позволяет правильнее понимать принципы функционирования и взаимодействия прикладной программы и операционной системы.

Пример рисования на окне с применением Windows API в Delphi.

Рассмотрим задачу рисования отрезка прямой линии на окне Windows.

1. Создайте новое обычное оконное приложение в Delphi.
2. Поместите на форму кнопку из палитры компонентов. По умолчанию Delphi даст ей имя Button1.
3. Дважды щелкните мышкой на кнопке Button1 и создайте, таким образом, процедуру обработки нажатия на кнопку.

Текст процедуры обработки сделайте в следующем виде:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   hdc: THandle;
begin
  hdc:= GetDC(Form1.Handle);
  LineTo(hdc,100,100);
  ReleaseDC(Form1.Handle,hdc);
      DeleteDC(hdc);
end;
```

В трех строках в этой процедуре используются три функции Windows API. Запустите программу на выполнение. После нажатия на кнопку на главном окне приложения будет нарисована диагональная линия, идущая из левого верхнего угла клиентской области окна. Рисование линии происходит с помощью функции *LineTo*, которая рисует линию из текущего положения рисования в точку с указанными координатами (x,y) на окне. Как ни странно может показаться на первый взгляд, первый параметр *hdc* в этой функции НЕ является непосредственной ссылкой на окно, в которое осуществляется графический вывод. Этот параметр называется **контекстом устройства** (Device Context)**.** Смысл его в том, что он является ссылкой на устройство вывода, которое, в данном случае, представляет собой окно. Под термином ссылка здесь подразумевается просто некоторый идентификатор, который операционная система присваивает устройству в ответ на запрос прикладной программы.

Почему наше окно операционная система называет устройством? Потому что рисование графики средствами операционной системы может производиться не только на окно монитора, но, например, на принтер или в область оперативной памяти для создания цифрового образа изображения. При этом для вывода графики прикладная программа пользуется одними и теми же функциями для рисования на устройствах, которые имеют разную физическую природу.

Для получения контекста устройства мы воспользовались функцией *GetDC*. На входе этой функции требуется указать идентификатор окна, с которым мы собираемся работать в дальнейшем. Мы использовали свойство формы *Form1.Handle,* которое представляет собой идентификатор окна для оконных компонентов Delphi, к каким относится форма. Этот параметр имеет тип *HWND*. Функция *GetDC* возвращает значение типа HDC, который в Delphi обозначается как *THandle*.

Определение. Под контекстом устройства подразумевается совокупность данных, описывающих параметры устройства, необходимые операционной системе для работы с ним.

Третья функция в нашей процедуре, *ReleaseDC*, выполняет освобождение оконного идентификатора от контекста устройства. Эта функция специально предназначена для оконных контекстов устройств, и первым из двух параметров у нее идет идентификатор окна. Последняя функция *DeleteDC(hdc)* удаляет контекст устройства и освобождает ресурсы операционной системы связанные с ним.

Рисовать с помощью функций Windows API можно также и на других оконных элементах Windows, например, на поверхности самой кнопки *Button1*, которая, с точки зрения операционной системы, является просто одним из окон. Для этого в процедуре надо вместо параметра *Form1.Handle* использовать *Button1.Handle*. Сам экран или поверхность рабочего стола Windows также имеет оконный идентификатор. Это означает, что для рисования на полном экране можно воспользоваться нашей процедурой. Для этого нужно подставить вместо параметра *Form1.Handle* значение *0*, которому равен идентификатор окна рабочего стола Windows.

Замечание. Подключение функций Windows API в приложениях Delphi делается с помощью интерфейсного модуля Windows.pas. В предыдущем примере нам не пришлось специально подключать этот модуль, так как предложение *uses,* которое генерируется автоматически, уже содержит указание на модуль Windows.pas. Можете проверить это самостоятельно, заглянув в текст модуля Unit1.pas из предыдущего примера.

Создание и отображение окна с использованием функций Windows API.

Создание окна и показ его на экране в Windows происходит в три этапа.

1. Регистрация класса окна с помощью функции *RegisterClass*;
2. Создание окна с помощью функции *CreateWindow*;
3. Показ окна на экране с помощью функции *ShowWindow*.

Далее рассмотрим пример, с помощью которого убедимся, что кнопка на окне Windows сама является окном. Для этого создадим необычную кнопку с характерной особенностью окна - строкой заголовка. Описание функции создания окна *CreateWindow* на языке программирования С выглядит немного устрашающе. Действительно, в документации имеется обширное описание параметров этой функции.

HWND CreateWindow(

```
    LPCTSTR lpClassName,        //указатель на зарегистрированное имя
                                //класса окна
     LPCTSTR lpWindowName,      //указатель на имя окна
     DWORD dwStyle,             //стиль окна
     int x,                     //расположение окна по горизонтали
     int y,                     //расположение окна по вертикали
     int nWidth,                //ширина окна
     int nHeight,               //высота окна
     HWND hWndParent,           //идентификатор (handle - хендл)
                                //родительского окна или окна-владельца
     HMENU hMenu,               //идентификатор оконного меню или
                                //дочернего окна
     HANDLE hInstance,          //идентификатор экземпляра приложения
     LPVOID lpParam             //указатель на данные для создания окна
```

```
);
```

Однако для создания стандартных элементов управления можно воспользоваться упрощенной схемой. Например, для создания кнопки можно указать предопределенный класс окна “BUTTON”, который нужно передать как строку, оканчивающуюся нулевым символом, в Object Pascal это тип *PChar*.

Для следующего примера добавьте на форму приложения еще одну кнопку. Система Delphi даст ей имя Button2. Придайте обработчику нажатия на кнопку Button2 следующий вид.

```
procedure TForm1.Button2Click(Sender: TObject);
var
    hw: HWND;
begin
hw:= CreateWindow(
  PChar('BUTTON'),
  PChar('MyWindow'),
        WS_VISIBLE,
        100,30,
        70,50,
        0,
        0,
        0,
        nil);
  //ShowWindow(hw,SW_SHOW);
 end;
```

При указании параметра *WS_VISIBLE* даже не обязательно вызывать функцию показа окна на экране. Оно будет автоматически видимым сразу после создания с помощью функции *CreateWindow*. Поэтому вызов функции *ShowWindow* специально показан закомментированным как необязательный.

В результате работы данной процедуры вы должны увидеть на экране кнопку, у которой имеется строка заголовка как у обычного окна с надписью MyWindow. Эта же надпись будет помещена и на самой кнопке. Кнопка будет существовать как отдельное окно на рабочем столе Windows.

Замечание. В данной теме не приводится исчерпывающего описания процесса создания окон Windows. Для подробного ознакомления с процессом создания и показа окон Windows обратитесь к документации или специальной литературе.

Каким же образом создать кнопку, чтобы она выглядела как обычно и принадлежала окну формы *Form1*? Для этого нужно при вызове функции CreateWindow изменить три параметра: *dwStyle*, *hWndParent* и *hInstance*. В параметре стиля окна указываем что оно является дочерним: *WS_CHILD*; указываем идентификатор окна-предка или окна-владельца: *Form1.Handle*; указываем уникальный идентификатор экземпляра приложения, который операционная система передает приложению сразу после запуска: *hInstance*. В Delphi для этого имеется специальная глобальная переменная, которая так и называется: *var HInstance: LongWord*. Эта переменная описана в модуле SysInit. Хотя в операционных системах Windows NT/2000/XP этот параметр при создании окна просто игнорируется.

Итак, вызов функции *CreateWindow* для создания кнопки на форме *Form1*:

```
hw:= CreateWindow(
```

```
PChar('BUTTON'),
PChar('MyWindow'),
WS_CHILD or WS_VISIBLE,
100,30,
85,50,
Form1.Handle,
0,
hInstance,
nil);
```

Рисование на окне Windows

Графическое изображение на окне создается с помощью объектов рисования линий и закрашивания. Эти объекты являются таковыми лишь по названию, как совокупность ресурсов операционной системы, а не в смысле объектно-ориентированного программирования. Линии имеют такие характеристики как толщина, вид, цвет. Закраска может иметь определенный цвет и тип, например, закраска горизонтальными, диагональными линиями, сплошная и т п. . При выводе на экран рисованной фигуры с помощью функции Windows API система определяет что в этой фигуре должно быть нарисовано с помощью линий, а что закрашено. Объект рисования линий называется Pen – перо, а объект для закраски Brush – кисть.

Для указания на определенный тип пера или кисти их нужно предварительно создать с помощью функции CreateObject. Затем, указать эти объекты в качестве текущих в контексте устройства окна в которое планируется осуществить вывод графики. Выбор текущего объекта осуществляется функцией SelectObject. Освобождение ресурсов операционной системы, связанных с объектами перо и кисть производится с помощью функции DeleteObject.

Параметрами функции SelectObject могут быть кроме пера и кисти также и некоторые другие объекты, которые перечислены ниже. Для создания объектов, которые используются при рисовании средствами Windows API, также могут применяться специализированные функции, использование которых может быть оправдано в некоторых типичных ситуациях.

Описание функции:

HGDIOBJ SelectObject(

```
        HDC hdc,                // идентификатор контекста устройства
        HGDIOBJ hgdiobj         //идентификатор выбираемого объекта
       );
```

В нашем случае выбора пера или кисти на выходе функции получаем идентификатор соответствующего объекта бывшего текущим до вызова функции *SelectObject*. Этот идентификатор объекта типа *HGDIOBJ* следует использовать для установления в качестве текущего после применения функций рисования с использованием определенного пера или кисти.

Параметр *hgdiobj* типа *HGDIOBJ* определяет выбираемый объект. Этот объект должен быть предварительно создан одной из следующих функций:

Объект	**Функции**
Pen	CreatePen, CreatePenIndirect

Brush	CreateBrushIndirect, CreateDIBPatternBrush, CreateDIBPatternBrushPt, CreateHatchBrush, CreatePatternBrush, CreateSolidBrush
Font	CreateFont, CreateFontIndirect
Bitmap	CreateBitmap, CreateBitmapIndirect, CreateCompatibleBitmap, CreateDIBitmap, CreateDIBSection
Region	CombineRgn, CreateEllipticRgn, CreateEllipticRgnIndirect, CreatePolygonRgn, CreateRectRgn, CreateRectRgnIndirect

Здесь *Font* – шрифт; *Bitmap* – битовая карта, точечное (растровое) изображение; *Region* – регион, область, определяющая определенную часть окна, которая должна быть видима на экране и соответственно невидимую область окна. С помощью регионов можно создать, например, кнопку овальной или другой произвольной формы. Для подробного ознакомления с данными объектами следует обратиться к документации.

Пример рисования на окне с использованием объектов пера и кисти

В этом примере мы изобразим на форме Form1 прямоугольник, граница у которого нарисована красной линией толщиной 2 пиксела, а закрашен прямоугольник зеленым цветом. Напомню, что пиксел – это точка или элемент растрового изображения на мониторе, происходит от английского “pixel” – picture element.

Поместите на форму кнопку. Этой третьей кнопке в нашем приложении Delphi даст название Button3. Создайте обработчик нажатия на эту кнопку в следующем виде:

```
procedure TForm1.Button3Click(Sender: TObject);
 var
 hdc       : THandle;
 hp,oldhp  : HPEN;
 hb,oldhb  : HBRUSH;
begin
     hdc:= GetDC(Form1.Handle);
 hp:= CreatePen(PS_SOLID,2,clRed);
 hb:= CreateSolidBrush(clGreen);
 oldhp:= SelectObject(hdc,hp);
 oldhb:= SelectObject(hdc,hb);
     Rectangle(hdc,10,10,150,80);
   SelectObject(hdc,oldhp);
   SelectObject(hdc,oldhb);
 deleteObject(hp);
 deleteObject(hb);
     ReleaseDC(Form1.Handle,hdc);
end;
```

В начале определяем контекст устройства: *hdc:= GetDC(Form1.Handle)*, который освобождается в конце процедуры. В функции CreatePen первым параметром указано, что линии должны быть сплошными: PS_SOLID, второй параметр говорит о том что толщина линии 2 пиксела, а для установки красного цвета линий использовалась цветовая константа Delphi *clRed*. При установке пера и кисти функция

SelectObject возвращает идентификатор текущих пера и кисти, которые сохраняются в переменных *oldhp* и *oldhb* соответственно. Это позволяет восстановить настройки контекста устройства по завершении операций рисования после вызова функции Rectangle с помощью двух вызовов функции *SelectObject*. Далее в процедуре объекты перо и кисть удаляются вызовом функции *deleteObject*. Конечно, если ресурсы компьютера позволяют, то создавать и удалять эти объекты можно в начале и конце работы программы соответственно, что ускорит работу приложения.

Глава 11. Избранные главы OpenGL. Введение.

Основные возможности OpenGL

OpenGL это графическая библиотека, которая содержит набор функций для работы с двумерной и трехмерной графикой. OpenGL является стандартной библиотекой в большинстве 32-х разрядных операционных системах. Она присутствует во всех версиях Windows. Это означает, что программы, использующие OpenGL, могут без больших усилий быть перенесены на разные платформы, что является одним из плюсов этой библиотеки.

OpenGL – универсальная, гибкая, популярная графическая библиотека. Она сочетает многочисленные современные достижения в области компьютерной графики с относительной легкостью изучения. OpenGL остается основным “конкурентом” для таких графических библиотек, как DirectX.

Библиотека OpenGL была разработана фирмой Silicon Graphics, Inc. (SGI) еще в эпоху структурного программирования. Поэтому ее применение в чем-то похоже на вызов функций Windows API. Методы объектно-ориентированного программирования в ней не применяются вовсе.

Итак, применение OpenGL сводится к описанию структур данных и вызову функций, которые обрабатывают эти данные.

В состав библиотеки входят три файла: opengl32.dll, glu32.dll, и glaux.dll. Первые два из этих файлов формируют основу библиотеки. Третий, glaux.dll, считается дополнительным. OpenGL может работать как на основе центрального процессора, так и с помощью аппаратного графического ускорителя, т.е. микросхем встроенных в плату видеоадаптера. Аппаратное ускорение для трехмерной графики осуществляется с помощью устанавливаемого клиентского драйвера (Installable Client Diver, ICD) и мини драйвера (Mini-Client Driver, MCD).

Здесь будут рассмотрены далеко не все возможности OpenGL. В основном темы курса посвящены вопросам настройки различных методов и параметров освещения и отражения света от поверхностей трехмерных объектов. От решения этой проблемы во многом зависит качество воспроизведения трехмерной сцены. Второй темой, на которой остановимся более подробно, будет наложение текстур на поверхности трехмерных объектов.

Вот некоторые из возможностей, которые включает в себя библиотека OpenGL:

- Рисование графических примитивов, таких как отрезки, треугольники, многоугольники, сферы, цилиндры на плоскости и в пространстве.
- Поддержка различных моделей источников света, таких как точечный, направленный и другие.
- Преобразования переноса, масштабирования и вращения трехмерных объектов.
- Изображение трехмерных объектов в параллельной и перспективной проекциях.
- Поддержка свойств материалов поверхностей.
- Сплайновые кривые и поверхности Безье и NURBS.
- Использование трафаретов.
- Эффект тумана.
- Смешение цветов и прозрачность.
- Создание тени и отражения.

- Текстурное наложение.
- Сечения трехмерных объектов.
- Вывод трехмерного текста на основе двумерных шрифтов Windows.

Также в OpenGL имеется возможность моделировать тени и отражение, однако для этого не предусмотрено встроенных средств, т.е. можно создать лишь имитацию тени и отражения.

Замечание. Библиотека OpenGL предназначена в первую очередь для создания интерактивных приложений компьютерной графики. Поэтому в ней не используется такой сложный метод как *обратная трассировка лучей*, который, несмотря на превосходные результаты при визуализации трехмерных сцен, все еще не может считаться достаточно быстродействующим.

Некоторые функции в OpenGL поддерживают специальную нотацию или форму записи, которая позволяет легче запоминать функции, которые выполняют схожие действия с входными параметрами разных типов.

Например, функция установки значения текущего цвета вершины *glColor* имеет 16 модификаций в зависимости от типа и количества входных параметров. В соответствии с [15] можно представить такие функции в общем виде:

rtype **CommandName**[1 2 3 4][b s i f d ub us ui][v] (atype arg*)*

Команда состоит из имени и трех символов, которые могут встречаться в различных комбинациях, хотя не все из них обязательно будут встречаться.

CommandName	Имя команды, например, *glColor*
[1 2 3 4]	Цифра, показывающая количество аргументов команды
[b s i f d ub us ui]	Символы, определяющие тип аргумента
[v]	Буква, показывающая что в качестве аргумента используется указатель на массив значений.

Рассмотрим типы данных которые используются в OpenGL.

Символ	Тип OpenGL	Соответствие в C
b	GLbyte	char
s	GLshort	short
i	GLint	int
f	GLfloat	float
d	GLdouble	double
ub	GLubyte	unsigned byte
us	GLushort	unsigned short
ui	GLuint	unsigned int

В качестве примера рассмотрим два вида вызова функции *glColor*:

```
glColor4f (0.8,0.5,0.4,1.0);
glColor3i (200,109,38);
```

Контекст воспроизведения

Рассмотрим процедуру *FormCreate*, которая выполняет подготовительные действия для работы с OpenGL. Главное, что здесь требуется – создание *контекста воспроизведения* OpenGL и установка его текущим. OpenGL недостаточно просто ссылки на окно Windows или контекст устройства. Контекст воспроизведения связывает окно с так называемым конвейером OpenGL с учетом установленного формата пикселов.

```
procedure TfrmGL.FormCreate(Sender: TObject);
  var
    pfd          : TPixelFormatDescriptor;
    nPixelFormat : integer;
begin
  hdc := GetDC (Panel1.Handle);
  FillChar(pfd,SizeOf(pfd),0);
  pfd.nSize:= SizeOf(pfd);
  pfd.dwFlags:=PFD_DOUBLEBUFFER;
  nPixelFormat:= ChoosePixelFormat(hdc,@pfd);
    if nPixelFormat = 0 then
    begin
       ShowMessage('Ошибка OpenGL');
       Halt;
    end;

  SetPixelFormat(hdc,nPixelFormat,@pfd);
  RContext :=  wglCreateContext(hdc);
  WglMakeCurrent(hrc,RContext);

  glClearColor (0.0, 0.0, 0.0,1.0); // цвет фона
  glEnable(GL_DEPTH_TEST);
  SphereObj:=gluNewQuadric;
  CylObj   :=gluNewQuadric;
  LightObj :=gluNewQuadric;
end;
```

Первый оператор в этой процедуре – получение оконного контекста устройства с помощью вызова функции GetDC. На основе полученного значения выбираем и устанавливаем формат пиксела. Формат пиксела – это структура на C или, как в данном случае, запись на ObjectPascal типа *TPixelFormatDescriptor*. В документации эта структура содержит следующие поля:

```
typedef struct tagPIXELFORMATDESCRIPTOR { // pfd

  WORD  nSize;//размер структуры данных
  WORD  nVersion;//версия структуры данных – всегда 1.
  DWORD dwFlags;//битовые флаги, например: PFD_SUPPORT_GDI
   BYTE  iPixelType;//тип пиксела: PFD_TYPE_RGBA или
 //PFD_TYPE_COLORINDEX
  BYTE  cColorBits;//битов для описания цвета пиксела, кроме Alpha
  BYTE  cRedBits;//битов красной компоненты пиксела
  BYTE  cRedShift;
  BYTE  cGreenBits; //битов зеленой компоненты пиксела
  BYTE  cGreenShift;
  BYTE  cBlueBits; //битов синей компоненты пиксела
```

```
BYTE  cBlueShift;
BYTE  cAlphaBits;
BYTE  cAlphaShift;
BYTE  cAccumBits;
BYTE  cAccumRedBits;
BYTE  cAccumGreenBits;
BYTE  cAccumBlueBits;
BYTE  cAccumAlphaBits;
BYTE  cDepthBits;//битов элемент глубины в z-буфере
BYTE  cStencilBits;
BYTE  cAuxBuffers;
BYTE  iLayerType;//для этой версии всегда PFD_MAIN_PLANE
BYTE  bReserved;
DWORD dwLayerMask;
DWORD dwVisibleMask;
DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

Для выбора требуемого формата пиксела необходимо заполнить наиболее важные поля записи и сделать запрос к OpenGL. Если система в состоянии работать с заданным форматом, то результат вызова функции *ChoosePixelFormat(hdc,@pfd)* будет ненулевым.

Для установки формата пиксела, который наиболее близок к тому, который установлен в операционной системе, достаточно обнулить все поля записи, и вызвать *ChoosePixelFormat*, предварительно указав в поле *pfd.nSize* размер самой записи в байтах. В качестве единственного требования указываем на необходимость двойной буферизации графического вывода: *pfd.dwFlags:=PFD_DOUBLEBUFFER*. При этом изображение будет строиться на невидимой области, а затем полностью выводиться на экран. Это гарантирует нам отсутствие мерцания изображения при смене кадров.

При успешном получении номера формата пикселов устанавливаем его:

*SetPixelFormat(hdc,*nPixelFormat*,@pfd).*

Далее, на основе контекста устройства и формата пикселов создаем контекст воспроизведения OpenGL:

RContext := wglCreateContext(hdc);
и устанавливаем его текущим:
wglMakeCurrent(hdc,RContext).

Для завершения работы с OpenGL требуется освободить текущий контекст устройства:

```
wglMakeCurrent(0, 0) ,
```

и затем удалить его:

```
wglDeleteContext(RContext).
```

Как видим, с помощью функции *wglMakeCurrent* можно не только устанавливать текущий контекст воспроизведения, но и освобождать его. Рассмотрим определение этой функции:

```
BOOL wglMakeCurrent(
   HDC   hdc, //Контекст устройства на который осуществляется вывод графики
   HGLRC  hglrc    //Контекст воспроизведения OpenGL для текущего потока
```

```
        );
```

Для освобождении контекста воспроизведения потока первый параметр игнорируется, а второй должен иметь нулевое значение. При успешном выполнении функция возвращает значение True.

Завершающие действия описаны в процедуре *FormDestroy*:

```
  procedure TfrmGL.FormDestroy(Sender: TObject);
  begin
    wglMakeCurrent(0, 0);
    wglDeleteContext(RContext);
    ReleaseDC (Handle, RContext);
    DeleteDC (hdc);
  end;
```

Параметры визуализации

Для изучения моделей отражения и освещения в OpenGL рассмотрим фрагменты программы, которая предназначена для интерактивной и наглядной демонстрации работы этих моделей.

Процедура вывода на экран очередного кадра изображения является ключевой. Для того чтобы автоматическая перерисовка окна не стирала изображение кадра, все действия встроены в обработчик перерисовки формы *FormPaint*.

```
procedure TfrmGL.FormPaint(Sender: TObject);
var
  ps : TPaintStruct;
  begin
   try
    BeginPaint (Panel1.Handle, ps); //для устойчивой работы
      //очистка буферов цвета и глубины
      glClear (GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT );
          SetUpCommons;
          DrawScene;
       glFlush;
       SwapBuffers(hrc);
    EndPaint (Panel1.Handle, ps);
   except
   end;
end;
```

Для большей надежности работы операторы заключены в конструкцию try..except, а также в программные скобки BeginPaint – EndPaint. Функция BeginPaint подготавливает окно для вывода графики.

Собственно обновление окна производится вызовом функций OpenGL:

- *glFlush;*
- *SwapBuffers(hrc);*

Функция glFlush заставляет сервер OpenGL закончить все команды по построении сцены OpenGL за конечное, как сказано в документации, время. Таким образом, мы

форсируем построение очередного кадра полностью прежде чем он будет показан на экране.

Функция SwapBuffers выводит изображение очередного кадра с невидимой области построения на экран. В качестве входного параметра этой функции указывается идентификатор контекста устройства, типа HDC, который был предварительно получен для окна Windows, в которое осуществляется вывод графики. Функция возвращает значение true при успешном выполнении.

За построение очередного кадра ответственны три функции:

- *glClear (GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);*
- *SetUpCommons;*
- *DrawScene;*

Функция *glClear* предназначена для очистки буферов OpenGL. Буферы OpenGL это двумерные массивы размером с окно вывода, которые содержат различные специфические параметры трехмерной сцены. Например, в нашем случае используется два буфера: буфер цвета и буфер глубины. Как можно догадаться, буфер цвета содержит информацию о цвете пикселов, а в буфере глубины содержится текущее значение “глубины” пикселов, т.е. значение координат точек по оси z, которые видимы наблюдателю. Битовая константа на входе функции glClear, которая ответственна за очистку буфера цвета, называется GL_COLOR_BUFFER_BIT, а буфера глубины - GL_DEPTH_BUFFER_BIT. В OpenGL используются и некоторые другие буферы, рассматривать их мы пока не будем.

Следующие две пользовательские функции интенсивно используют функции OpenGL. Функция SetUpCommons предназначена для установки необходимых параметров визуализации. Внутри функции DrawScene строятся трехмерные объекты и осуществляются их перемещения и другие необходимые трансформации с учетом предварительно установленных параметров визуализации.

Рассмотрим каким образом настраиваются параметры вида и проекции в функции SetUpCommons:

```
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity;

 Case RGProj.ItemIndex of
   0:  glOrtho   (-LC,LC,-Panel1.height/Panel1.width*LC,
                  Panel1.height/Panel1.width*LC, 3, 1000);

   1: glFrustum (-1,1,-Panel1.height/Panel1.width,
    Panel1.height/Panel1.width, 3, 1000);
end;//case

procedure TfrmGL.InitViewPort;
   begin
    glViewport(0, 0, Panel1.Width, Panel1.Height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity;
    glFrustum (-1,1,-Panel1.height/Panel1.width,
                     Panel1.height/Panel1.width, 3, 1000);
end;
```

Функция glViewport устанавливает так называемый порт вывода. Порт вывода это прямоугольная область в пределах окна Windows, в которое осуществляется отображение графики. Иногда порт вывода так и называют в английской транскрипции - “вьюпорт”:

```
void glViewport(
GLint x,
GLint y,
GLsizei width,
GLsizei height
);
```

Параметры *x* и *y* определяют левый нижний угол порта вывода в пикселах. Если порт вывода явно не указан, то (x,y)=(0,0). Параметры *width* и *height* задают ширину и высоту порта вывода, соответственно. При первом подключении контекста воспроизведения размеры порта вывода совпадают с размерами окна Windows к которому подключен контекст воспроизведения.

Команда *glMatrixMode(GL_PROJECTION)* устанавливает матрицу проекций текущей. В OpenGL используется общеизвестная модель матричных преобразований трехмерной графики, которая включает в себя матрицу видового преобразования, и матрицу проекции. Матрица видового преобразования задает расположение, перемещения, и масштабирование объектов в пространстве, а также расположение объектов по отношению к наблюдателю. Матрица проекции определяет параметры проецирования на экран. В OpenGL можно установить любую матрицу перспективного преобразования непосредственно. Но для параллельной и центральной перспективной проекций можно воспользоваться специализированными функциями.

После установки текущей матрицы она содержит в себе значения, которые в общем случае могут быть не известны. Если применять преобразования, связанные с этой матрицей, то текущие значения будут использоваться, что может привести к непредсказуемым результатам на экране. Для того чтобы этого не произошло, текущую матрицу следует инициализировать как единичную с помощью функции *glLoadIdentity*, либо запомнить ее в стеке для последующего восстановления.

Из двух типов проекций рассмотрим в начале центральную перспективную проекцию. Она устанавливается командой `glFrustum`:

```
void glFrustum(
GLdouble left,
GLdouble right,
Gldouble bottom,
GLdouble top,
GLdouble near,
GLdouble far
);
```

Параметры этой функции определяют координаты расположения граней усеченной пирамиды видимого объема: left и right – для левой и правой граней относительно ближней *near* грани. Нижняя и верхняя грани соответствуют параметрам *bottom* и *top*, а дальняя отсекающая плоскость определяется параметром *far*. Обратите внимание, что в приведенном примере параметры левой и правой стороны усеченной пирамиды кажутся выбранными слишком малыми по сравнению с параметрами верхней и нижней

сторон. Однако это вполне допустимо, поскольку OpenGL самостоятельно подстраивает пропорции видимого объема под параметры окна отображения Windows.

В библиотеке glu32.dll также находится функция установки перспективной проекции *gluPerspective*, однако ее мы рассматривать не будем.

Глава 12. Модели освещенности граней трехмерных объектов в OpenGL

На следующем рисунке представлена слегка упрощенная схема трех моделей освещенности граней в OpenGL.

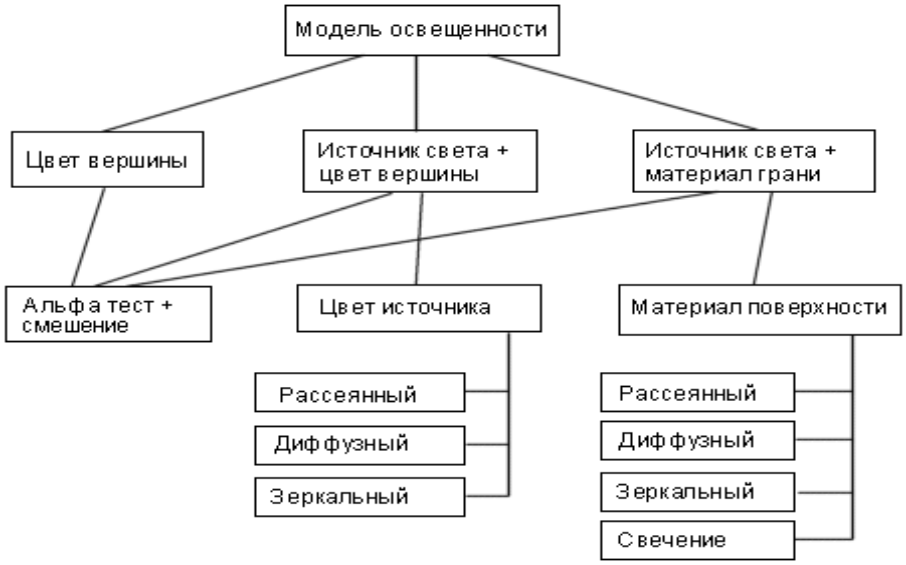

Рис. 244 Схема моделей освещенности в OpenGL.

Всего рассмотрим три модели освещенности:

1. Модель, в которой используется только цвет вершины, который приписывается каждой вершине при ее создании.
2. Модель, в которой цвет вершины взаимодействует с цветом источника света.
3. Модель, в которой цвет источника света взаимодействует со свойствами материала поверхности. Цвет вершины при этом игнорируется.

Материалом поверхности называется совокупность ее отражающих характеристик. Они воздействуют на цвет отраженных компонент падающего на поверхность света.

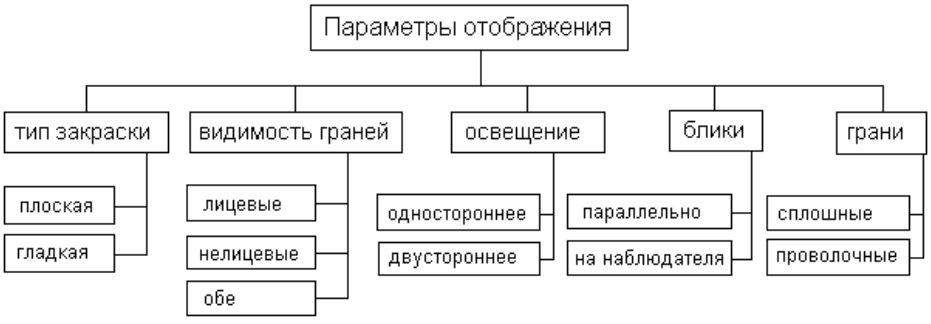

Рис. 255 Параметры отображения в OpenGL

В приведенной схеме моделей освещенности показаны только основные факторы, влияющие на *цвет* результирующих поверхностей. Но существуют и другие параметры, от которых зависит внешний вид граней объектов трехмерной сцены. На

следующей схеме показаны параметры, от настройки которых зависит *способ отображения* граней и их совокупностей в трехмерной сцене. Некоторые из них имеют смысл при оптимизации и ускорении вывода графики, или для придания большей реалистичности внешнему виду трехмерных объектов.

Модель освещенности и параметры отображения объектов необходимо устанавливать перед созданием трехмерных объектов, поскольку результат построения объекта сразу же заносится в соответствующие буферы OpenGL. Поэтому прежде чем создавать объект нужно быть уверенным, что все настройки освещенности и отображения для него установлены правильно.

Процедура *SetUpCommons*, о которой упоминалось выше, как раз устанавливает эти настройки.

Модель освещенности с использованием цвета вершины

Эта модель требует минимальных усилий как со стороны OpenGL, так и со стороны программиста. Рассмотрим процедуру, в которой создается сфера.

```
procedure TfrmGL.DrawSphere;
begin
  glColor4f(0.8,0.5,0.4,SphereAlpha.Position/100);//текущий цвет
     Case RGSphere.ItemIndex of
          0: begin
               gluQuadricDrawStyle (SphereObj, GLU_FILL);
             end;
          1: begin
               gluQuadricDrawStyle (SphereObj, GLU_LINE);
             end;
     end;//case
   gluSphere(SphereObj, 35, 25, 25 );
end;
```

Функция *glColor4f* устанавливает текущий цвет которым будут изображены объекты. Первые три параметра указывают на значения компонент цветов RGB, последний параметр соответствует значению альфа-канала цвета вершин. *Альфа канал* это числовой параметр, который используется при дополнительной обработке цвета вершины при визуализации. Часто его используют для осуществления эффекта прозрачности или полупрозрачности. Значения всех четырех параметров функции *glColor4f* принимают значения от 0 до 1. Для альфа канала, в случае эффекта прозрачности, значение 1 соответствует полной непрозначности. Параметр уровня прозрачности ползунка *SphereAlpha.Position* может принимать значения от 0 до 100, что при делении на 100 дает правильный диапазон для альфа канала цвета вершин сферы.

В нашей процедуре сфера создается с помощью *Quadric-объектов* библиотеки OpenGL. Многие графические примитивы можно достаточно легко создать как Quadric-объекты. Это позволяет программисту не определять каждую вершину трехмерного объекта, а создать его полностью всего лишь несколькими командами OpenGL.

Создание сферы начинается еще при создании формы в обработчике *FormCreate*:

```
var
SphereObj: PGLUquadricObj;
```

```
Begin
 . . .
   SphereObj:= gluNewQuadric;
 . . .
end;
```

Затем, для формирования каждого нового кадра изображения трехмерной сцены нужно указывать, каким именно должен быть Quadric-объект. В операторе *case* указываем способ построения сферы:

как сплошную:

```
gluQuadricDrawStyle (SphereObj, GLU_FILL),
```

или в виде проволочной модели:

```
gluQuadricDrawStyle (SphereObj, GLU_LINE).
```

После предварительных настроек даем команду на отрисовку сферы: *gluSphere(SphereObj, 35, 25, 25)*. Числовые параметры в этой функции есть, соответственно, радиус сферы, количество делений по меридиану или вдоль оси z, (которая проходит в направлении, перпендикулярном плоскости экрана), и количество делений “по широте”, по аналогии с географическим делением земного шара.

Следует учитывать, что при применении модели освещенности с использованием цвета вершины источники света должны быть отключены. Это достигается с помощью одной команды: *glDisable(GL_LIGHTING)*.

Здесь мы встречаемся с парой часто используемых функций OpenGL:

```
void glEnable(GLenum cap) и
void glDisable(GLenum cap),
```

- так выглядит их описание на языке C в документации.

Эти функции предназначены, соответственно, для включения и выключения определенных настроек визуализации OpenGL. Параметр *cap* определяет символическую константу параметра визуализации.

Например, включение режима освещения достигается вызовом функции *glEnable(GL_LIGHTING).*

Отображение одноцветных трехмерных объектов в данном режиме сильно затрудняет восприятие трехмерности за счет того, что грани одного цвета сливаются в одну цветовую область на экране. При повототах и перемещениях объектов цвет не изменяется, так что грани всегда выглядят как пятно одного цвета.

Замечание. Данный режим освещенности в комбинации с режимом смешения цветов можно с успехом применить, например, для имитации тени.

Получение эффекта полупрозрачности

Задать полупрозрачность можно с помощью функции смешения цветов. Этот эффект получается когда новое изображение накладывается поверх старого. В этом случае цвета пикселов в одинаковых позициях представляются аргументами функции смешения с учетом их альфа-компонент.

В начале включаем режим смешения цветов:

```
glEnable(GL_BLEND);
```

Затем устанавливаем требуемую функцию смешения, которая дает необходимый эффект:

```
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
```

Функция *glBlendFunc* принимает два параметра, первый из которых определяет функцию, применяемую к цвету-источнику, а второй функцию применяемую к цвету который накладывается поверх текущего. В документации по OpenGL указывается, что наилучший эффект полупрозрачности достигается сочетанием параметров, которые приведены в данном примере. Эффект наиболее хорошо выглядит если выводить многоугольники в порядке от наиболее удаленного к наименее удаленному. Функция *glBlendFunc* может принимать ряд других значений своих параметров, их подробный список можно найти в документации.

Отключается режим смешения цветов командой:

```
glEnable(GL_BLEND).
```

Модель освещенности с использованием источника света и цвета вершины

В этой модели освещенности рассмотрим свойства источников света в OpenGL. Система OpenGL версии 1 поддерживает всего 8 или менее одновременно действующих источников света. В общем случае для выяснения максимально разрешенного количества источников света используем функцию *glGetIntegerv(GL_MAX_LIGHTS, @num)*, где *num* – целочисленная переменная, в которую запишется результат - количество источников света.

Источники света задаются именованными константами вида `GL_LIGHTx,` где вместо x подставляется номер источника, начиная с нулевого. Например, включение первого источника минимально достигается вызовом двух команд:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Источник света в OpenGL состоит из нескольких цветовых составляющих: рассеянный свет – Ambient, диффузный – Diffuse, зеркальный – Specular. Хотя кажется что это противоречит здравому смыслу, но в OpenGL вполне можно настроить источник света так, например, чтобы он “светил” красной рассеянной составляющей, зеленой диффузной составляющей и синей зеркальной. В результате можно получить очень интересные визуальные эффекты. Рассмотрим фрагмент программы настройки компонент источника света с помощью векторной версии функции *glLight*:

```
var
//параметры источника света
ambient: array[1..4]of GLfloat = (0.7, 0.9, 0.9, 1.0);
diffuse: array[1..4]of GLfloat = (1.0, 1.0, 1.0, 1.0);
specular: array[1..4]of GLfloat = ( 1.0, 1.0, 1.0, 1.0 );
begin
...
glLightfv(GL_LIGHT0, GL_AMBIENT,  @ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE,  @diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, @Specular);
```

Функция *glLight* может использоваться и для установок некоторых других параметров источника света.

В качестве значений цветовых составляющих используются четырехкомпонентные массивы RGBA. Первый вызов *glLight* в нашем примере задает цвет рассеянной составляющей. Если не указывать ее явно, то по умолчанию система считает что рассеянный свет отсутствует: (0.0, 0.0, 0.0,1.0).

Второй вызов устанавливает диффузную составляющую, которая для нулевого источника по умолчанию равняется белому цвету (1.0, 1.0, 1.0, 1.0), а для остальных – черному, т.е. отсутствует: (0.0, 0.0, 0.0, 1.0). Третий вызов *glLightfv* с параметром *GL_SPECULAR* определяет зеркальную составляющую. По умолчанию интенсивность зеркальной составляющей для нулевого источника равняется (1.0, 1.0, 1.0, 1.0), а для остальных (0.0, 0.0, 0.0,1.0).

Другие параметры функции *glLight* позволяют настраивать такие характеристики источника света как его направленность, положение в пространстве, формулу ослабления интенсивности в зависимости от удаления от поверхности, распределение светового пятна. По умолчанию создается точечный источник с равномерным распределением интенсивности, без ослабления в зависимости от удаленности.

Как видно из рис. 44, на модель освещенности с использованием источника света и цвета вершины также влияют установки смешения цветов, рассмотренные выше.

На рис. 46 показан пример освещения сферы точечным источником света в модели освещенности с использованием источника света и цвета вершины.

Рис. 266. Освещение сферы точечным источником света в модели освещенности с использованием источника света и цвета вершины.

Модель освещенности с использованием источника света и материала поверхности

Включение режима достигается командой:

```
glEnable(GL_COLOR_MATERIAL),
```

а выключение:

```
glDisable(GL_COLOR_MATERIAL).
```

Материал поверности в OpenGL определяет векторы коэффициентов отражения компонент источника света от поверхности: рассеянной, диффузной и зеркальной, а также самосвечение. Вектор каждого из этих трех компонентов равен цвету, который получится при освещении поверхности белой компонентой источника света.

Рассмотрим пример задания материала поверхности.

```
glMaterialfv(GL_FRONT,GL_AMBIENT,@fr_Mat_Amb);
glMaterialfv(GL_FRONT,GL_DIFFUSE,@fr_Mat_Diff);
glMaterialfv(GL_FRONT,GL_SPECULAR,@fr_Mat_Spec);
glMaterialfv(GL_FRONT,GL_EMISSION,@fr_Mat_Emit);
glMaterialfv(GL_FRONT,GL_SHININESS,@fr_Mat_Shine);
```

В OpenGL можно задать материал как для лицевой так и для обратной стороны поверхности в зависимости от значения первого параметра функции *glMaterial*: *GL_FRONT* или *GL_BACK*.

Глава 13. Параметры отображения в OpenGL

Под параметрами отображения будем понимать различные настройки изображения трехмерных объектов в OpenGL, которые не связаны непосредственно с цветовыми характеристиками поверхностей. При перечислении параметров отображения будем руководствоваться рис. 45, который был приведен выше.

Итак, рассмотрим пять параметров, которые могут использоваться в большинстве приложений трехмерной графики.

- Тип закраски
- Видимость граней
- Освещение граней
- Блики
- Изображение граней

Тип закраски: плоская или гладкая

Поверхность трехмерных объектов обычно моделируется с помощью многоугольников, так что для получения гладких поверхностей пришлось бы уменьшать размеры этих многоугольников, что привело бы к росту их количества и, соответственно, к резкому увеличению времени необходимого для расчета очередного кадра изображения. Однако проблему получения гладких поверхностей можно решить за счет визуального размывания стыков граней поверхностей с помощью математических методов. Эта возможность реализована в OpenGL. Функция *glShadeModel* позволяет отображать грани плоскими, какими они есть или сгладить стыки между ними, создавая иллюзию гладкой поверхности. В случае сглаживания цвет пикселов поверхности получается за счет интерполяции цветов соседних вершин.

```
glShadeModel(GL_FLAT);//плоские грани
glShadeModel(GL_SMOOTH);//сглаживание поверхности
```

Закономерности отражения света от поверхности учитывают направление нормали к поверхности, которое программист может задавать произвольно. (По умолчанию направление нормали определяется перпендикулярно в сторону лицевой плоскости грани, а лицевая плоскость задается обходом вершин грани против часовой стрелки.)

Видимость граней: лицевые, нелицевые

Грани поверхностей имеют две стороны. Одна из них называется лицевой, другая – нелицевой. По умолчанию, лицевая поверхность задается обходом вершин грани против часовой стрелки. Часто поверхности трехмерных объектов обращены к пользователю только одной стороной. Такая ситуация типична, например, для замкнутых объектов, внутренность которых никогда не может быть увидена. Эту особенность удобно использовать для повышения скорости построения кадра. Например, если грань оказалась повернутой к пользователю нелицевой стороной, то ее можно просто отбросить и не учитывать при расчете кадра изображения.

Учет показа разных сторон граней осуществляется включением с помощью символьной константы *GL_CULL_FACE*:

`glEnable(GL_CULL_FACE)` – включает режим показа разных сторон граней.

В случае выключения режима учета сторон граней будут отображаться обе стороны граней: `glDisable(GL_CULL_FACE)`.

Выключение видимости лицевой или нелицевой сторон осуществляется функцией *glCullFace*, которая может принимать один из двух входных параметров:

`glCullFace(GL_BACK)` – выключает отображение нелицевых сторон граней.
`glCullFace(GL_FRONT)` - выключает отображение лицевых сторон граней.

При желании можно задать способ обхода вершин граней – по часовой стрелке или против часовой стрелки для определения лицевой поверхности грани. Для этого используется функция *glFrontFace*.

Вариант *glFrontFace(GL_CW)* – задает обход по часовой стрелке, а *glFrontFace(GL_CCW)* – против часовой стрелки (Counter ClockWise).

Освещение: одностороннее или двустороннее

В случае двустороннего отображения граней в OpenGL можно задать необходимость использования материала нелицевых граней. Иначе нелицевые грани изображаются без учета освещения с использованием их цвета вершин.

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 1)`

– включить использование материала нелицевых граней.

`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 0)`

– нелицевые грани отображаются цветом вершин без учета освещения. По умолчанию значение второго параметра равняется нулю.

Расчет бликов: параллельно или с учетом положения наблюдателя

Блики на поверхностях получаются в модели зеркального отражения. Расчет бликов в OpenGL осуществляется не по реальным физическим законам, а с применением математических моделей, которые создают визуальный эффект, который похож на настоящий, но с меньшими вычислительными затратами.

Имеется два способа отображения бликов.

`glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, 0)`

– в этом случае направление на наблюдателя считается параллельным оси (*-z*) текущей видовой системы координат для любой точки на поверхности трехмерных объектов. Эта модель задает облегченную схему вычислений. Такие настройки применяются по умолчанию.

```
glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, 1)
```

– в этом случае при расчете бликов учитывается реальное направление на место расположения наблюдателя. Эта схема дает более реалистичную картину, но требует больше вычислений по сравнению с первым случаем.

Грани: сплошные или проволочные

Для Quadric-объектов способ отображения граней уже рассматривался в главе, посвещенной модели освещенности с использованием цвета вершины.

`gluQuadricDrawStyle (SphereObj, GLU_FILL)` – для сплошных граней
`gluQuadricDrawStyle (SphereObj, GLU_LINE)` – для проволочных граней;

В общем же случае, то есть при создании трехмерных объектов с помощью *примитивов* OpenGL, перед указанием координат вершин граней следует установить тип текущего примитива. Для получения проволочной модели следует использовать режим рисования линий, и сопутствующие этому режиму параметры:

```
glBegin(GL_LINES).
```

Для вывода сплошных граней следует воспользоваться режимами рисования треугольников, четырехугольников или других полигонов. Однако есть команда, которая даже в режиме рисования многоугольников позволяет выбирать как они должны изображаться при растеризации. Это команда

```
glPolygonMode(
   GLEnum face,
   GLEnum mode)
```

Параметр *mode* задает режим отображения и может принимать следующие значения [15]:

GL_POINT	В этом режиме изображаются только отдельные вершины многоугольника, для которых установлен флаг грани. Для этого режима действуеи режим устранения ступенчатости *GL_POINT_SMOOTH*.
GL_LINE	Стороны многоугольников изображаются в виде отрезков. У них можно изменять ширину (*GL_LINE_WIDTH*) и устанавливать режим устранения ступенчатости (*GL_LINE_SMOOTH*).
GL_FILL	Внутренняя область многоугольника закрашивается текущим цветом. В этом режиме также может быть применен режим устранения ступенчатости (*GL_POLYGON_SMOOTH*).

Параметр *face* определяет многоугольники, к которым применяется режим, заданный параметром *mode*:

`GL_FRONT` – лицевые многоугольники;
`GL_BACK` – нелицевые многоугольники;
`GL_FRONT_AND_BACK` – лицевые и нелицевые.

Рассмоотрим теперь полный текст процедуры установки параметров визуализации SetupCommpns.

```
procedure TfrmGL.SetUpCommons;
const
LC = 53.0;
begin
glMatrixMode(GL_PROJECTION);
glLoadIdentity;

Case RGProj.ItemIndex of
 0:  glOrtho   (-LC,LC,-Panel1.height/Panel1.width*LC,
                Panel1.height/Panel1.width*LC, 3, 1000);


1: glFrustum (-1,1,-Panel1.height/Panel1.width,
     Panel1.height/Panel1.width, 3, 1000);     //задаем перспективу
     end;//case
//Альфа тест
if CBAlphaTest.Checked then
     glEnable(GL_ALPHA_TEST)
else
     glDisable(GL_ALPHA_TEST);
//Блендинг
if CBBlending.Checked then
      begin
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
     end
else
     glDisable(GL_BLEND);
//закраска
Case RGFlatSm.ItemIndex of
0: glShadeModel(GL_FLAT);
1: glShadeModel(GL_SMOOTH);
      end;//case
Case RGVisEdges.ItemIndex of
0:begin
  glEnable(GL_CULL_FACE);
  glCullFace(GL_BACK);
      end;
1:begin
  glEnable(GL_CULL_FACE);
  glCullFace(GL_FRONT);
     end;
2:begin glDisable(GL_CULL_FACE)end;
     end;//case
//Освещение
```

```
if CBColorMaterial.Checked then//учет свойств материала
     glEnable(GL_COLOR_MATERIAL)
else
     glDisable(GL_COLOR_MATERIAL);
if  CBoxLightOn.Checked then
  begin
  //включаем свет
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
      glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @lmodel_ambient);
  //источники данных для параметров освещения
  glLightfv(GL_LIGHT0, GL_AMBIENT, @ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, @Specular);
  end
  else
  begin
    glDisable(GL_LIGHTING);
    glDisable(GL_LIGHT0);
      end;
//способ зеркальной освещенности
if CBoxLockViewer.Checked then
     //зеркально - в направлении наблюдателя, а не параллельно
     glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, @glTrue) //1.0
     else
glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, @glFalse);//0.0
//двусторонний/односторонний рендеринг
if CBoxDoubSided.Checked then
  //не 0 - двусторонний рендеринг
  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 1)
     else
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 0);
//Материал объектов сцены
  glMaterialfv(GL_FRONT,GL_AMBIENT,@fr_Mat_Amb);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,@fr_Mat_Diff);
  glMaterialfv(GL_FRONT,GL_SHININESS,@fr_Mat_Shine);
  glMaterialfv(GL_FRONT,GL_SPECULAR,@fr_Mat_Spec);
  glMaterialfv(GL_FRONT,GL_EMISSION,@fr_Mat_Emit);

  glMaterialfv(GL_BACK,GL_AMBIENT,@back_Mat_Amb);
  glMaterialfv(GL_BACK,GL_DIFFUSE,@back_Mat_Diff);
  glMaterialfv(GL_BACK,GL_SHININESS,@back_Mat_Shine);
  glMaterialfv(GL_BACK,GL_SPECULAR,@back_Mat_Spec);
  glMaterialfv(GL_BACK,GL_EMISSION,@back_Mat_Emit);

end;//procedure TfrmGL.SetUpCommons;
```

Глава 14. Пространственные геометрические преобразования в OpenGL

Вершины объектов, а также матрицы преобразований в OpenGL используют однородные координаты. То есть вершины объектов являются 4-х компонентными векторами, а матрицы преобразований имеют размер 4x4.

На рис.47 показаны три объекта в пространстве: проволочная сфера, полый цилиндр внутри нее и источник света. Пусть нам требуется поворачивать эти объекты относительно центра сферы. Поворот будем осуществлять с помощью мыши, нажатием и перемещением указателя в нужном направлении.

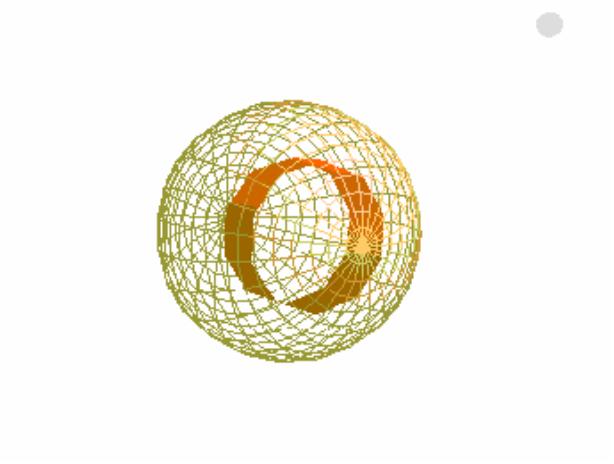

Рис. 277. Проволочная сфера, цилиндр внутри нее и источник света.

Для перемещений в пространстве используется текущая матрица в режиме GL_MODELVIEW. Эту матрицу можно сформировать непосредственно, но можно воспользоваться функциями, задающими преобразования привычным для нас способом. Одного и того же эффекта можно добиться перемещением объектов относительно наблюдателя, а также перемещением наблюдателя относительно объектов в пространстве. Поэтому важно правильно понимать действие функций преобразования, которые включают в себя функции переноса, поворота и масштабирования.

Рассмотрим процедуру построения объектов *DrawScene*, изображенных на рис. 47.

```
procedure TfrmGL.DrawScene;
begin
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
        glTranslatef(0,0,-260);//задаем матрицу переноса
        glRotatef(AngleX, 0.0, 1.0, 0.0); // поворот вокруг оси Y
        glRotatef(AngleY, 1.0, 0.0, 0.0); // поворот вокруг оси X
  DrawLight;
  DrawCylinder;
  DrawSphere;
 end;
```

После установления текущей единичной матрицы видового преобразования *GL_MODELVIEW* следуют три функции, которые задают расположение и текущий угол поворота сферы, цилиндра и источника света относительно центра сферы. Последующие команды *DrawLight*,*DrawCylinder* и *DrawSphere* создают источник света, цилиндр и сферу, к которым автоматически применяется матрица видового преобразования.

Как было замечено выше, последовательность функций преобразования координат можно рассматривать применительно как к вершинам трехмерных объектов, так и к системе координат наблюдателя. Это может привести к некоторой путанице. К сожалению, в литературе и документации по OpenGL объяснение действия функций преобразования координат явно недостаточное, так что изучающие вынуждены приходить к пониманию с помощью многочисленных проб и экспериментов. Попытаемся составить понятную картину происходящего. Для этого не будем углубляться в какие-либо математические выкладки на языке матриц и векторов. Укажем лишь фактический алгоритм, которым руководствуется система OpenGL.

Во-первых, необходимо указать на то, что после установления режима *GL_MODELVIEW* мы будем иметь дело не с одной только матрицей видового преобразования. В реальности мы всегда имеем взаимодействие двух систем координат. Одна из них – *мировая* система координат – описывает расположение всех объектов в пространстве, в том числе и точку расположения, и ориентацию системы координат наблюдателя. Вторая из них – система координат наблюдателя или *видовая* система координат. По умолчанию начала этих систем координат совпадают. Совпадают также их оси *x* и *y*. Однако оси *z* этих систем координат не совпадают по направлению.

Мировая система координат традиционно является правосторонней, а видовая система координат левосторонней. Такое расположение осей систем координат связано с интуитивным и привычным представлением об их расположении. Например, начало видовой системы координат расположено в центре экрана монитора, ось *x* направлена слева направо, ось *y* снизу вверх, а ось *z* – перпендикулярно экрану в направлении от пользователя к монитору. Это дает левостороннюю систему. Однако при изображении объектов в трехмерном пространстве удобнее рисовать и представлять систему координат как правостороннюю. Как раз то же самое, что и в мировой системе координат.

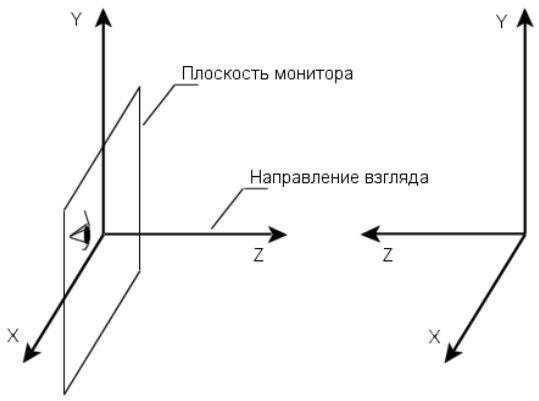

Рис. 28. Расположение видовой (слева) и мировой (справа) систем координат. Мировая система координат смещена вправо относительно своего первоначального положения.

Далее приводятся два правила, которыми следует руководствоваться для мысленного представления результата действия функций геометрических преобразований OpenGL.

В начале рассмотрим случай, когда считаем, что функции преобразований применяются к координатам вершин трехмерных объектов. В этом случае взаимное

расположение мировой и видовой систем координат остается неизменным. Изменяются только координаты объектов относительно ***мировой*** системы координат. Действие функций геометрических преобразований при этом следует учитывать в порядке ***обратном*** тому, в котором они записаны в тексте программы.

Во втором случае, мы считаем, что действие функций геометрических преобразований относится к ***видовой*** системе координат, то есть к осям системы координат наблюдателя относительно начального положения ***видовой*** же системы координат. В этом случае мировая система координат остается неподвижной вместе с объектами, который в ней описаны. Задавая координаты объектов, мы каждый раз переносим наблюдателя и располагаем ***оси*** его системы координат в соответствии с текущей матрицей преобразования, которая была задана с помощью функций *glTranslate*, *glRotate* и *glScale*. Действие функций геометрических преобразований при этом следует учитывать в том порядке, в котором они записаны в тексте программы.

Для проверки правильности этих двух правил применим их на примере функции *DrawScene*, приведенной выше. Итак, матрица геометрических преобразований составлена всего из трех действий:

```
glTranslatef(0,0,-260);//задаем матрицу переноса
glRotatef(AngleX, 0.0, 1.0, 0.0); // поворот вокруг оси Y
glRotatef(AngleY, 1.0, 0.0, 0.0); // поворот вокруг оси X
```

Прежде чем представлять себе результат действия последовательности этих функций следует выбрать одно из двух вышеприведенных правил. Пусть, например, это первое правило, по которому преобразования применяются к координатам вершин объектов в мировой системе координат. Тогда следует рассматривать команды преобразований (но не изменять текст программы) в обратном порядке:

```
glRotatef(AngleY, 1.0, 0.0, 0.0); // поворот вокруг оси X
glRotatef(AngleX, 0.0, 1.0, 0.0); // поворот вокруг оси Y
glTranslatef(0,0,-260);//задаем матрицу переноса
```

Функции поворота задают вращение объектов относительно начала координат мировой системы. Оси систем обеих систем координат при этом остаются на своих местах. Первая команда поворота вращает объекты вокруг оси вращения образованной вектором (1,0,0), то есть вокруг оси OX мировой системы координат. Положительное вращение, т.е. поворот на положительный угол, происходит при повороте против часовой стрелки, если смотреть из положительной полуоси образованной вектором вращения (в данном случае из положительной полуоси OX) в направлении начала координат. Поскольку центры сферы и других объектов совпадают с началом координат, то объекты пока остаются на своих местах. Третья функция *glTranslatef(0,0,-260)* переносит объекты на (–260) единиц по оси Z в мировой системе координат. Все объекты удаляются от начала координат мировой системы, а значит и от начала координат видовой системы координат на 260 единиц. Вспомним, что отрицательное направление оси Z мировой системы координат соответствует положительному направлению видовой системы координат. Поэтому на экране мы видим, что объекты перенесены на (+260) единиц вперед.

Рассмотрим второй вариант, когда преобразуется система координат наблюдателя относительно мировой системы, а объекты остаются неподвижными. В этом случае последовательность команд остается той же что и в тексте программы. Первым действием выполняется перенос начала координат наблюдателя на (-260) единиц по оси

Z системы координат наблюдателя. Это дает эффект как если бы мы отодвинулись от монитора назад. Затем производится поворот вокруг оси Y первоначального положения видовой системы координат, а после этого и вокруг оси X тоже первоначального положения видовой системы координат. Поскольку ось Z видовой системы координат всегда остается направленной на начало координат первоначальной видовой системы и на начало координат мировой системы, то получается эффект как будто мы облетаем начало мировой системы координат, глядя на него. Но это неотличимо от того, как если бы объекты вращались перед нами, оставаясь все время на одном месте!

Поскольку при вращении видовой системы координат взгляд остается направленным на центр первоначальной видовой системы координат, то неявно задается и вращение новой системы координат наблюдателя относительно ее центра. При этом постоянным остается угол между осями текущей системы координат наблюдателя и перпендикуляром, опущенным из начала текущей системы координат наблюдателя на ось вращения в первоначальной видовой системе координат. Наглядно этот поворот можно представить как скольжение системы координат наблюдателя по окружности основания конуса, образующая которого равна расстоянию от наблюдателя до центра первоначальной видовой системы координат.

Таким образом, показано, что два указанных правила приводят к одним и тем же визуальным результатам. Ими можно пользоваться в зависимости от того хотим ли мы представить перемещения объектов в мировом пространстве, или перемещения наблюдателя в мировом пространстве, то есть относительно мировой системы координат.

Рассмотрим далее описания некоторых других функций, которые участвуют в процессе формирования матрицы видового преобразования в OpenGL.

Для операции масштабирования объектов относительно начала координат применяется функция glScale, которая существует в двух версиях: glScaled и glScalef. В качестве параметров они принимают значения масштабных коэффициентов по трем координатным осям. Напомним, что значение масштабного коэффициента равное единице не изменяет размеров объекта вдоль данной оси.

Описание функции glScalef:

```
void glScalef(
GLfloat x,
GLfloat y,
GLfloat z
);
```

Если геометрические преобразования разных объектов должны производится не в заранее оговоренной последовательности, а асинхронно, как, например, это может произойти при использовании разных программных потоков, вывод которых осуществляется в один и тот же контекст воспроизведения, то перед установкой новой матрицы видового преобразования следует запомнить ее текущие значения. Для этого текущая матрица запоминается в стеке OpenGL с помощью функции *glPushMatrix*. Восстановление сохраненной матрицы достигается вызовом функции *glPopMatrix*. Обе эти функции используются без входных параметров.

Глава 15. Наложение текстур в OpenGL

Текстура в OpenGL это рисунок или битовая карта (Bitmap), которая накладывается на грани поверхностей трехмерных объектов. Текстуры создают иллюзию рельефа на поверхности и придают объектам соответствующую раскраску. Например, если на поверхность сферы наложить изображение карты мира, то получится модель земного шара. Если же рисунок изменить на чередующиеся черные и зеленые полосы, то сфера уже будет больше похожа на арбуз. Безусловно, текстуры играют важнейшую роль при моделировании трехмерных объектов.

Размеры текстуры в OpenGL по ширине и высоте в пикселах должны быть степенью двух. Это требование связано с применением оптимизированных алгоритмов рисования текстур при разложении их в растр. Максимальный размер текстуры ограничен. В каждой реализации OpenGL его можно узнать вызовом функции

```
  glGetIntegerv(GL_MAX_TEXTURE_SIZE,@GLMaxTex),
```

где переменная *GLMaxTex* типа *GLInt* после вызова будет содержать значение максимального размера текстуры в пикселах. Разрешение и запрещение наложения двумерных текстур достигается командами *glEnable* и *glDisable* с параметром *GL_TEXTURE_2D*.

Загрузка образа текстуры

Для загрузки битовой карты в качестве текущей текстуры используется функция *glTexImage2D*. Эта функция загружает образ битовой карты в оперативную память с помощью указателя. Прежде чем изучать параметры этой функции рассмотрим текст пользовательской функции *LoadBmpTexture*, написанной на Object Pascal, которая загружает текстуру из файла на диске.

```
function LoadBmpTexture(xSize,ySize: GLInt;Name: string):
pointer;
type
 TRGB = record
          r,g,b: GLUByte;
        end;
 PBits = ^TBits;
 TBits = Array [0..0] of GLUbyte;
var
  i, j: Integer;
  bitmap: TBitmap;
  Size: GLInt;
  Bits: PBits;
begin
   bitmap := TBitmap.Create;
   bitmap.LoadFromFile(Name); // загружаем текстуру из файла
   Size:= xSize*ySize*SizeOf(TRGB);
   GetMem(Bits,Size);

// заполнение битового массива
 For i:= 0 to xSize-1 do
```

```
  For j:= 0 to ySize-1 do
  begin
    bits[(i*xSize+j)*3+0]:=
GetRValue(bitmap.Canvas.Pixels[j,xSize-1-i]);
    bits[(i*xSize+j)*3+1]:=
GetGValue(bitmap.Canvas.Pixels[j,xSize-1-i]);
    bits[(i*xSize+j)*3+2]:=
GetBValue(bitmap.Canvas.Pixels[j,xSize-1-i]);
  end;
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                xSize, ySize,// здесь задается размер текстуры
                0, GL_RGB, GL_UNSIGNED_BYTE, bits);
 Result:=bits;
 //FreeMem(bits);
 bitmap.Free;
end;//function LoadBmpTexture
```

Как видим, внутри функции *LoadBmpTexture* вызывается функция загрузки текстуры *glTexImage2D*. Особенность загрузки текстур состоит в том, что в образе битовой карты информация о пикселах хранится не в виде R-G-B, как это принято в формате изображений DIB, а наоборот B-G-R, то есть первой идет компонента синего цвета, затем зеленого и далее красного. Удобство функции *LoadBmpTexture* также состоит в том, что метод загрузки картинки из файла объекта *TBitmap* автоматически преобразует любые форматы DDB в универсальный формат DIB, так что нам остается только правильно поменять местами расположение байтов тройки RGB, что и происходит в двойном цикле:

```
 For i:= 0 to xSize-1 do
  For j:= 0 to ySize-1 do
  begin
    bits[(i*xSize+j)*3+0]:=
GetRValue(bitmap.Canvas.Pixels[j,xSize-1-i]);
    bits[(i*xSize+j)*3+1]:=
GetGValue(bitmap.Canvas.Pixels[j,xSize-1-i]);
    bits[(i*xSize+j)*3+2]:=
GetBValue(bitmap.Canvas.Pixels[j,xSize-1-i]);
  end;
```

Рассмотрим параметры функции *glTexImage2D*, как она использована в программе.

```
glTexImage2D(  GL_TEXTURE_2D, 0, GL_RGBA,
           xSize, ySize, 0, GL_RGB,
           GL_UNSIGNED_BYTE, bits
          );
```

Первый параметр в этой функции может принимать только значение символической константы *GL_TEXTURE_2D*. Второй параметр указывает значение уровня детализации текстуры. Уровень детализации задает уменьшение текстуры в заданное число раз с применением алгоритмов Mipmap. По умолчанию используется значение 0. Третий параметр задает количество цветовых компонент текстуры, может быть от 1 до 4. Четвертый и пятый параметры задают ширину и высоту текстуры. Ширина должна удовлетворять выражению 2^n+2 x Border, где Border = 0 или 1, ширина границы текстуры, задается в следующем, шестом параметре. Седьмой параметр задает формат пикселов, может принимать следующие значения: *GL_COLOR_INDEX, GL_RED,*

GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE, GL_LUMINANCE_ALPHA. Предпоследний, восьмой параметр определяет значение типа данных для каждой пиксельной компоненты: *GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT*. Последний параметр представляет собой указатель на начало области данных текстуры. В нашем случае это переменная *bits*, описана как указатель на массив байт. Возможно, вас смутил тип данных

```
TBits = Array [0..0] of GLUbyte;
```

Действительно, такое описание выглядит как ошибка в программе. Однако, это сделано специально. Данная функция должна компилироваться с выключенной опцией на проверку допустимых диапазонов {$R-}. Тогда блок памяти *bits* в программе мы можем рассматривать как массив байт произвольной длины, и использовать обычный синтаксис Object Pascal для доступа к его элементам. Конечно, контроль выхода за пределы области памяти такого массива мы должны полностью брать на себя.

После загрузки образа текстуры в память следует указать грань или грани трехмерных объектов, на которые будет осуществляться наложение текстуры. Здесь мы подходим к определению *координат текстуры*. Необходимо установить соответствие между вершинами граней трехмерного объекта и местом на текстуре, которое каждой вершине соответствует. Начало системы координат текстуры расположено в левом нижнем углу прямоугольного рисунка текстуры. Ось s, подобно оси абсцисс направлена слева направо по нижней кромке рисунка, а ось t, аналогично оси ординат направлена снизу вверх. Границы прямоугольника изображения текстуры задают диапазон изменения значений координат *s* и *t* от 0 до 1.

Рассмотрим пример создания прямоугольника в трехмерном пространстве и наложение на него изображения корабля из файла *Ship.bmp* размером 64x64 пиксела. Здесь используется возможность создания так называемых *дисплейных списков* OpenGL, которые позволяют объединить длинные последовательности команд OpenGL под одним названием и запускать их на выполнение. Дисплейные списки позволяют между командами OpenGL выполнять и другие операторы, но, в отличие от обычных процедур и функций, запоминаются в списке только команды OpenGL. При последующем выполнении команд дисплейного списка обычные операторы не выполняются.

```
Const
     Quad: GLInt=1;//идентификатор дисплейного списка
     Ship: pointer;
. . .
glNewList(Quad,GL_COMPILE);//создаем новый дисплейный список
 Ship:= LoadBmpTexture(64,64,'Ship.bmp');//загружаем текстуру в
//память
    glBegin (GL_QUADS);//задаем тип примитива - четырехугольники
        glTexCoord2d (0.0, 0.0);//левый нижний угол текстуры
        glVertex3f (-8.0, -8.0, 15.0);
     glTexCoord2d (1.0, 0.0); //правый нижний угол текстуры
        glVertex3f (8.0, -8.0, 22.0);
     glTexCoord2d (1.0,0.8);//верхняя часть изображения будет слегка
//приплюснута
```

```
        glVertex3f (8.0, 0.0, 15.0);
        glTexCoord2d (0.0, 1.0);//левый верхний угол текстуры
        glVertex3f (-8.0, 8.0, 15.0);
    glEnd;
glEndList;//конец создания списка
```

Рисование с использованием дисплейного списка происходит с помощью команды *glCallList(Quad)*.

В случае Quadric объектов рассмотрим пример наложения изображения корабля на сферу.

```
//устанавливаем изображение корабля в качестве текущей текстуры
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                64, 64, //размеры текстуры
                0, GL_RGB, GL_UNSIGNED_BYTE, Ship);

  glEnable(GL_TEXTURE_2D);//разрешаем использование текстур
  gluQuadricTexture(Sphere,GL_TRUE);//разрешаем наложение текстуры на
  //объект Sphere

   gluQuadricDrawStyle (Sphere, GLU_FILL);//сплошная закраска сферы
   gluSphere(Sphere, 15.0, 24, 24 );//рисуем сферу с наложением
//текстуры
```

Рисунок проецируется на сферу аналогично тому, как прямоугольная карта земного шара “заворачивала” бы глобус. То есть верхняя и нижняя кромки изображения текстуры после проецирования на сферу оказываются стянутыми в точку.

Параметры наложения текстуры

Из параметров визуализации рассмотрим установку параметров окружения и фильтрации пикселов. Параметры окружения устанавливаются функцией *glTexEnv[if]*. Действие ее состоит в формировании функции преобразования цветов источника света, цвета образа текстуры, цвета вершин примитивов и цвета конфигурации текстуры для получения результирующего цвета поверхности с наложенной на нее текстурой. Функция существует в двух вариантах в зависимости от входных параметров. Рассмотрим описание для вещественных входов:

```
void glTexEnvfv(
GLenum target,
GLenum pname,
const GLfloat *params
);
```

Параметр *target* всегда принимает значение *GL_TEXTURE_ENV*. Параметр *pname* задает один из двух вариантов *GL_TEXTURE_ENV_MODE* или *GL_TEXTURE_ENV_COLOR*.

В случае когда *pname= GL_TEXTURE_ENV_MODE* доступны три варианта вызова функции:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL),
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND),
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE).
```

Когда *pname=GL_TEXTURE_ENV_COLOR*, то третий параметр представляет собой указатель на четырехкомпонентный массив цвета:

```
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, @CColor),
```

где переменная описана как

```
var
  CColor: array[1..4]of Glfloat;
```

По умолчанию для этих команд установлены значения: *GL_MODULATE* для режима *GL_TEXTURE_ENV_MODE* и цвет (0,0,0,0) для режима *GL_TEXTURE_ENV_COLOR*.

Далее в таблице, в соответствии с [15], приводятся варианты формирования результирующей функции преобразования с помощью команды *glTexEnv*.

Количество цветовых компонент	**GL_MODULATE**	**GL_DECAL**	**GL_BLEND**
1	$C_v=L_tC_f$	нет	$C_v=(1-L_t)C_f+L_tC_c$ $A_v=A_f$
2	$C_v=L_tC_f$ $A_v=A_fA_t$	Нет	$C_v=(1-L_t)C_f+L_tC_c$ $A_v=A_fA_t$
3	$C_v=C_tC_f$ A=A	$C_v=C_t$ $A_v=A_f$	нет
4	$C_v=C_tC_f$ $A_v=A_fA_t$	$C_v=(1-A_t)C_f+A_tC_t$ $A_v=A_f$	нет

Здесь A – значение альфа канала; L – яркость. Для двухкомпонентных изображений цвет состоит из компонент яркости L и альфа составляющей A. Трехкомпонентный цвет в качестве яркости использует три цветовые компоненты C, а четырехкомпонентный цвет использует C и A.

Индексы в формулах: f – фрагмент, на который накладывается текстура, t – образ текстуры, c – цвет конфигурации текстуры, v – результирующее значение.

Из практических экспериментов можно заметить, что для обычных матовых поверхностей хороший визуальный эффект получается использованием режима GL_MODULATE. Простой режим отображения текстуры с эффектом

полупрозрачности можно получить в режиме GL_DECAL. В режиме GL_BLEND можно получить эффект негатива изображения.

Параметры масштабирования текстуры задаются функцией *glTexParameter[if]*.

Самый простой и быстрый но не самый реалистичный способ отображения при масштабировании задается комбинацией параметров в режиме GL_NEAREST:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST) –
```

для масштабирования в сторону увеличения текстуры, и

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST) -
```

для масштабирования в сторону уменьшения. Более приемлемые результаты с точки зрения сглаживания цветов соседних пикселов при масштабировании текстуры дает вариант линейного сглаживания:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
```

и

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
```

В режиме *GL_LINEAR* цвет пиксела текстуры получается как среднее арифметическое четырех соседних пикселов. Также возможны режимы с использованием битовых карт уровней детализации текстуры, которые задаются функцией *gluBuild2DMipmaps*.

Список литературы

1. Аммерал Л. Машинная графика на языке С. В 4-х томах – М: Сол. Систем, 1992.
2. Боресков А.Б., Шикина Е.В., Шикина Г.Е., Компьютерная графика: первое знакомство. – М.: Финансы и статистика, 1996.
3. Вэн-Дэм А., Фоли Дж. Основы интерактивной машинной графики.Т.1-2 – М.: Мир, 1985.
4. Гилой, Интерактивная машинная графика. - М.: Мир, 1981.
5. Грайс Графические средства персональных компьютеров. – М.: Мир, 1980.
6. Жикин Е.В., Боресков А.В. Компьютерная графика. Динамика, реалистические изображения. – М.: Диалог-МИФИ, 1995, 1997.
7. Компьютер обретает разум. Пер. с англ. Под ред. В.Л.Стефанюка – М.: Мир, 1990.
8. Ласло М. Вычислительная геометрия и компьютерная графика на С++. М.: Бином, 1997.
9. М.Маров, 3D Studio MAX 2.5: справочник – СПб: «Питер», 1999. – 672 с.
10. А.Ла Мот, Д.Ратклифф и др. Секреты программирования игр/ Перев с англ. – СПб: Питер, 1995. – 720 с.
11. Ньюмен, Спрулл, Основы интерактивной машинной графики. - М.: Мир, 1976.
12. Ф. Препарата, М. Шеймос, Вычислительная геометрия: Введение. - М. Мир, 1989.
13. Роджерс Алгоритмические основы машинной графики. – М.: Мир, 1989.
14. Роджерс, Адамс, Математические основы машинной графики. - М. Машиностроение, 1985.
15. Ю.Тихомиров, Программирование трехмерной графики - С.-Пб.: БХВ-Санкт-Петербург,1999.
16. Н. Томпсон, Секреты программирования трехмерной графики для Windows 95. Перев с англ. – СПб: Питер, 1997. – 352 с.
17. А.Фокс, М. Пратт, Вычислительная геометрия - М., Мир, 1982.
18. А.В.Фролов, Г.В.Фролов, Графический интерфейс GDI в MS WINDOWS – М.: Диалог-МИФИ, 1994.
19. Хонич А. Как самому создать трехмерную игру. – М.: МИКРОАРТ, 1996.
20. Энджел Й. Практическое введение в машинную графику. – М.: Радио и Связь, 1984.
21. А.Б.Боресков, Е.В.Шикина, Г.Е.Шикина, Компьютерная графика: первое знакомство, Под ред. Е.В.Шикина - М.: Финансы и статистика, 1996.
22. Краснов М, OpenGL. Программирование трехмерной графики на Delphi. – СПб.: БХВ – Петербург, 2000.
23. Краснов М., DirectX. Графика в проектах Delphi. – СПб.: БХВ – Петербург, 2001.
24. А.Ла Мот, Д.Ратклифф и др. Секреты программирования игр/ Перев с англ. – СПб: Питер, 1995. – 720 с.
25. М.Маров, 3D Studio MAX 2.5: справочник – СПб: «Питер», 1999. – 672 с.
26. Ю.Тихомиров, Программирование трехмерной графики. - С.-Пб.: БХВ-Санкт-Петербург,1999.

27. Н. Томпсон, Секреты программирования трехмерной графики для Windows 95. Перев с англ. – СПб: Питер, 1997. – 352 с.

28. Стен Трухильо, Графика для Windows средствами Direct Draw. - С.-Пб: Питер Ком, 1998 – 320 с.

29. А.В.Фролов, Г.В.Фролов, Графический интерфейс GDI в MS WINDOWS, Москва, Изд-во Диалог-МИФИ, 1994Майкл Ласло, Вычислительная геометрия и компьютерная графика на C++, - М.: Бином, 1997.

30. А.Хонич, Как самому создать трехмерную игру. - М.: МИКРОАРТ, 1996.

31. Джим Адамс, DirectX: продвинутая анимация. – М.: КУДИЦ-ОБРАЗ, 2004.

32. Пол Стид, Анимация персонажей для игр в реальном времени. – М.: ДМК Пресс, М, 2004.