МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ М.В. ЛОМОНОСОВА

ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И КИБЕРНЕТИКИ

ЛАБОРАТОРИЯ КОМПЬЮТЕРНОЙ
ГРАФИКИ И МУЛЬТИМЕДИА

Ю.М. Баяковский, А.В. Игнатенко, А.И. Фролов

Графическая библиотека OpenGL

учебно-методическое пособие

Москва
2003

УДК 681.3.07
ББК 32.973.26-018.2
Б34

Баяковский Ю.М., Игнатенко А.В., Фролов А.И. **Графическая библиотека OpenGL**. Учебно-методическое пособие.

Методическое пособие представляет собой практическое руководство по работе с графической библиотекой OpenGL. Оно включает описание базовых возможностей OpenGL и приемы работы с библиотекой. Рассматриваются вопросы оптимизации приложений. Пособие рассчитано на читателей, знакомых с языками программирования C/C++ и имеющих представление о базовых алгоритмах компьютерной графики. Рекомендуется студентам, аспирантам, научным сотрудникам.

Рецензенты:

Шикин Е.В., профессор, д.ф.-м.н.

Крылов А.С., к.ф.-м.н.

Издательский отдел факультета Вычислительной Математики и Кибернетики МГУ им. Ломоносова (лицензия НД № 05899 от 24.09.2001), 2003 г. – 132 с.

Печатается по решению Редакционно-Издательского Совета факультета Вычислительной Математики и Кибернетики Московского Государственного Университета им. М.В. Ломоносова.

© Факультет Вычислительной Математики и Кибернетики МГУ им. Ломоносова, 2003

© Лаборатория Компьютерной Графики и Мультимедиа, 2003

ISBN 5-89407-153-4

Содержание

Предисловие *6*

Введение *8*

Глава 1 Основы OpenGL *10*

1.1. Основные возможности **10**

1.2. Интерфейс OpenGL **11**

1.3. Архитектура OpenGL **12**

1.4. Синтаксис команд **14**

1.5. Пример приложения **15**

Контрольные вопросы: **19**

Глава 2 Рисование геометрических объектов *20*

2.1. Процесс обновления изображения **20**

2.2. Вершины и примитивы **21**

2.3. Операторные скобки glBegin / glEnd **23**

2.4. Дисплейные списки **28**

2.5. Массивы вершин **29**

Контрольные вопросы **31**

Глава 3 Преобразования объектов *32*

3.1. Работа с матрицами **32**

3.2. Модельно-видовые преобразования **35**

3.3. Проекции **36**

3.4. Область вывода **38**

Контрольные вопросы **39**

Глава 4 Материалы и освещение *40*

4.1. Модель освещения **40**

4.2. Спецификация материалов 41
4.3. Описание источников света 43
4.4. Создание эффекта тумана 46
Контрольные вопросы 48
Глава 5 Текстурирование 49
5.1. Подготовка текстуры 49
5.2. Наложение текстуры на объекты 52
5.3. Текстурные координаты 55
Контрольные вопросы 57
Глава 6 Операции с пикселями 58
6.1. Смешивание изображений. Прозрачность 59
6.2. Буфер-накопитель 61
6.3. Буфер маски 62
6.4. Управление растеризацией 64
Контрольные вопросы 65
Глава 7 Приемы работы с OpenGL 66
7.1. Устранение ступенчатости 66
7.2. Построение теней 67
7.3. Зеркальные отражения 72
Контрольные вопросы 75
Глава 8 Оптимизация программ 76
8.1. Организация приложения 76
8.2. Оптимизация вызовов OpenGL 80
Контрольные вопросы 88
Приложение A. Структура GLUT-приложения 89
Приложение B. Примитивы библиотек GLU и GLUT 93
Приложение C. Настройка приложений OpenGL 97

C.1. Создание приложения в среде Borland C++ 5.02 97
C.2. Создание приложения в среде MS Visual C++ 6.0 98
C.3. Создание приложения в среде Borland C++ Builder 6. 99
Приложение D. Демонстрационные программы 100
D.1. Пример 1: Простое GLUT-приложение 100
D.2. Пример 2: Модель освещения OpenGL 103
D.3. Загрузка BMP файла 106
D.4. Пример 3: Текстурирование 111
Приложение E. Примеры практических заданий 118
E.1. Cornell Box 118
E.2. Виртуальные часы 120
E.3. Интерактивный ландшафт 121
Литература 127
Предметный указатель 128

Предисловие

Мы стали свидетелями драматических изменений, которые произошли в компьютерной графике в 90-е годы. Если в конце 80-х графические рабочие станции стоили безумно дорого и работать с ними могли только в очень богатых организациях (как правило из ВПК), то в конце 90-х графические станции с вполне удовлетворительными возможностями за 1000 USD стали доступны университетам и даже отдельным студентам. Если в 80-е использовалась преимущественно векторная графика, то в конце 90-х растровая полноцветная графика почти полностью вытеснила векторную. Трехмерная графика стала столь же распространенной как двумерная, поскольку появились и быстро совершенствуются видеоплаты с графическими ускорителями и z-буфером.

Параллельно с изменениями графической аппаратуры происходили глубокие метаморфозы в программном обеспечении. Вслед за широким распространением в 70-е годы графических библиотек (в основном векторных, в большинстве своем фортранных) в 80-е годы потребовалось несколько этапов стандартизации графического обеспечения (Core System, PHIGS, GKS), чтобы к середине 90-х прийти к Открытой Графической Библиотеке (OpenGL). В настоящее время многие функции этой библиотеки реализованы аппаратно.

Все эти процессы не могли не сказаться на преподавании компьютерной графики в университетах. В 80-е годы и в первой половине 90-х целью курса было изучение и программирование базовых алгоритмов графики (рисование прямой и кривой, клиппирование, штриховка или растеризация многоугольника, однородные координаты и аффинные преобразования, видовые преобразования) [1,2]. Теперь, при наличии интерфейса прикладного программиста (API) высокого уровня, когда элементарные функции имеются в библиотеке OpenGL и зачастую реализуются аппаратно, пришлось пересмотреть концепцию курса. (В самом деле, зачем учиться умножать столбиком, если у каждого в руках калькулятор.) Появилась возможность включить в курс более сложные и более современные разделы компьютерной графики, такие как текстурирование, анимация. Именно в соответствии с этой общемировой тенденцией

эволюционировал курс компьютерной графики на факультете ВМиК МГУ [3,4].

Следуя принципу "учись, делая" (learning-by-doing), мы, кроме традиционных лекций, включаем в курс выполнение 5-6 небольших проектов, каждый продолжительностью две недели. (Примеры таких заданий вы найдете в этом пособии.) Настоящее пособие призвано помочь студентам в выполнении этих проектов. В отличие от других справочных публикаций по OpenGL, в пособии говориться не о том, что имеется в библиотеке, а о том, как этими средствами эффективно пользоваться. Например, как визуализировать зеркальные объекты, как построить тени. Пособие существует в электронном виде в течение четырех лет на сайте Лаборатории Компьютерной Графики и Мультимедиа (http://graphics.cs.msu.su), и все эти годы оно эволюционирует с учетом потребностей курса.

Авторы благодарны К. Дмитриеву, А. Куликовой и А. Дегтяревой, которые прочитали рукопись и сделали ценные замечания.

Ю.М.Баяковский

Декабрь 2002 года

Введение

OpenGL является одним из самых популярных прикладных программных интерфейсов (API – Application Programming Interface) для разработки приложений в области двумерной и трехмерной графики.

Стандарт OpenGL (Open Graphics Library – открытая графическая библиотека) был разработан и утвержден в 1992 году ведущими фирмами в области разработки программного обеспечения как эффективный аппаратно-независимый интерфейс, пригодный для реализации на различных платформах. Основой стандарта стала библиотека IRIS GL, разработанная фирмой Silicon Graphics Inc.

Библиотека насчитывает около 120 различных команд, которые программист использует для задания объектов и операций, необходимых для написания интерактивных графических приложений.

На сегодняшний день графическая система OpenGL поддерживается большинством производителей аппаратных и программных платформ. Эта система доступна тем, кто работает в среде Windows, пользователям компьютеров Apple. Свободно распространяемые коды системы Mesa (пакет API на базе OpenGL) можно компилировать в большинстве операционных систем, в том числе в Linux.

Характерными особенностями OpenGL, которые обеспечили распространение и развитие этого графического стандарта, являются:

- *Стабильность.* Дополнения и изменения в стандарте реализуются таким образом, чтобы сохранить совместимость с разработанным ранее программным обеспечением.
- *Надежность и переносимость.* Приложения, использующие OpenGL, гарантируют одинаковый визуальный результат вне зависимости от типа используемой операционной системы и организации отображения информации. Кроме того, эти приложения могут выполняться как на персональных компьютерах, так и на рабочих станциях и суперкомпьютерах.
- *Легкость применения.* Стандарт OpenGL имеет продуманную структуру и интуитивно понятный интерфейс, что позволяет с меньшими затратами создавать эффективные приложения,

содержащие меньше строк кода, чем с использованием других графических библиотек. Необходимые функции для обеспечения совместимости с различным оборудованием реализованы на уровне библиотеки и значительно упрощают разработку приложений.

Наличие хорошего базового пакета для работы с трехмерными приложениями упрощает понимание студентами ключевых тем курса компьютерной графики – моделирование трехмерных объектов, закрашивание, текстурирование, анимацию и т.д. Широкие функциональные возможности OpenGL служат хорошим фундаментом для изложения теоретических и практических аспектов предмета.

Глава 1

Основы OpenGL

1.1. Основные возможности

Описывать возможности OpenGL мы будем через функции его библиотеки. Все функции можно разделить на пять категорий:

- *Функции описания примитивов* определяют объекты нижнего уровня иерархии (примитивы), которые способна отображать графическая подсистема. В OpenGL в качестве примитивов выступают точки, линии, многоугольники и т.д.
- *Функции описания источников света* служат для описания положения и параметров источников света, расположенных в трехмерной сцене.
- *Функции задания атрибутов*. С помощью задания атрибутов программист определяет, *как* будут выглядеть на экране отображаемые объекты. Другими словами, если с помощью примитивов определяется, *что* появится на экране, то атрибуты определяют *способ* вывода на экран. В качестве атрибутов OpenGL позволяет задавать цвет, характеристики материала, текстуры, параметры освещения.
- *Функции визуализации* позволяет задать положение наблюдателя в виртуальном пространстве, параметры объектива камеры. Зная эти параметры, система сможет не только правильно построить изображение, но и отсечь объекты, оказавшиеся вне поля зрения.
- Набор *функций геометрических преобразований* позволяют программисту выполнять различные преобразования объектов – поворот, перенос, масштабирование.

При этом OpenGL может выполнять дополнительные операции, такие как использование сплайнов для построения линий и

поверхностей, удаление невидимых фрагментов изображений, работа с изображениями на уровне пикселей и т.д.

1.2. Интерфейс OpenGL

OpenGL состоит из набора библиотек. Все базовые функции хранятся в основной библиотеке, для обозначения которой в дальнейшем мы будем использовать аббревиатуру *GL*. Помимо основной, OpenGL включает в себя несколько дополнительных библиотек.

Первая из них – *библиотека утилит GL*(*GLU – GL Utility*). Все функции этой библиотеки определены через базовые функции GL. В состав GLU вошла реализация более сложных функций, таких как набор популярных геометрических примитивов (куб, шар, цилиндр, диск), функции построения сплайнов, реализация дополнительных операций над матрицами и т.п.

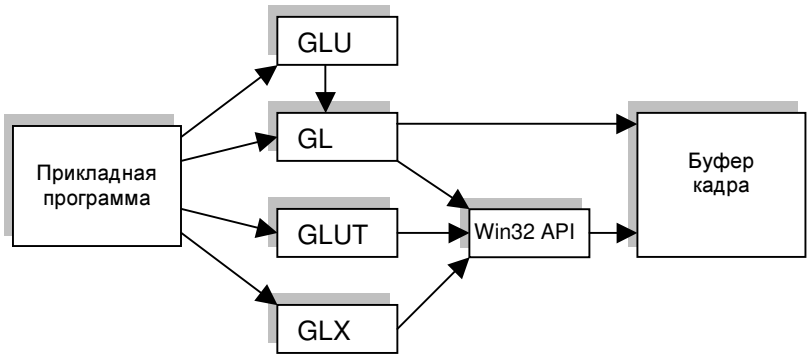


Рис. 1 Организация библиотеки OpenGL

OpenGL не включает в себя никаких специальных команд для работы с окнами или ввода информации от пользователя. Поэтому были созданы специальные переносимые библиотеки для обеспечения часто используемых функций взаимодействия с пользователем и для отображения информации с помощью оконной подсистемы. Наиболее популярной является библиотека GLUT (GL Utility Toolkit). Формально GLUT не входит в OpenGL, но de facto включается почти во все его дистрибутивы и имеет реализации для различных платформ. GLUT предоставляет только минимально необходимый набор функций для создания OpenGL-приложения. Функционально аналогичная

библиотека GLX менее популярна. В дальнейшем в этом пособии в качестве основной будет рассматриваться GLUT.

Кроме того, функции, специфичные для конкретной оконной подсистемы, обычно входят в ее прикладной программный интерфейс. Так, функции, поддерживающие выполнение OpenGL, есть в составе Win32 API и X Window. На рисунке схематически представлена организация системы библиотек в версии, работающей под управлением системы Windows. Аналогичная организация используется и в других версиях OpenGL.

1.3. Архитектура OpenGL

Функции OpenGL реализованы в модели клиент-сервер. Приложение выступает в роли клиента – оно вырабатывает команды, а сервер OpenGL интерпретирует и выполняет их. Сам сервер может находиться как на том же компьютере, на котором находится клиент (например, в виде динамически загружаемой библиотеки – DLL), так и на другом (при этом может быть использован специальный протокол передачи данных между машинами).

GL обрабатывает и рисует в буфере кадра графические *примитивы* с учетом некоторого числа выбранных режимов. Каждый примитив – это точка, отрезок, многоугольник и т.д. Каждый режим может быть изменен независимо от других. Определение примитивов, выбор режимов и другие операции описываются с помощью *команд* в форме вызовов функций прикладной библиотеки.

Примитивы определяются набором из одной или более *вершин* (vertex). Вершина определяет точку, конец отрезка или угол многоугольника. С каждой вершиной ассоциируются некоторые данные (координаты, цвет, нормаль, текстурные координаты и т.д.), называемые *атрибутами*. В подавляющем большинстве случаев каждая вершина обрабатывается независимо от других.

С точки зрения архитектуры графическая система OpenGL является конвейером, состоящим из нескольких последовательных этапов обработки графических данных.

Команды OpenGL всегда обрабатываются в том порядке, в котором они поступают, хотя могут происходить задержки перед тем, как проявится эффект от их выполнения. В большинстве случаев OpenGL предоставляет непосредственный интерфейс, т.е. определение объекта вызывает его визуализацию в буфере кадра.

С точки зрения разработчиков, OpenGL – это набор команд, которые управляют использованием графической аппаратуры. Если аппаратура состоит только из адресуемого буфера кадра, тогда OpenGL должен быть реализован полностью с использованием ресурсов центрального процессора. Обычно графическая аппаратура предоставляет различные уровни ускорения: от аппаратной реализации вывода линий и многоугольников до изощренных графических процессоров с поддержкой различных операций над геометрическими данными.

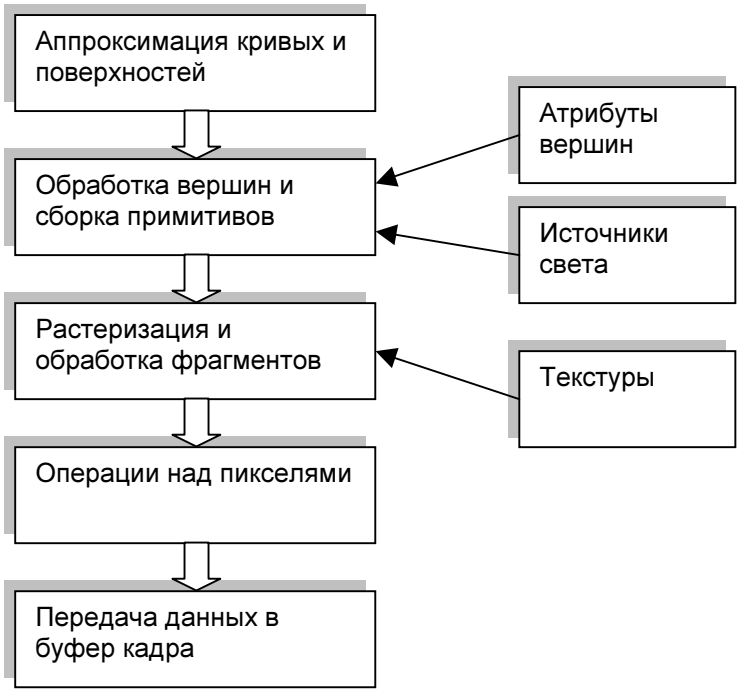


Рис. 2. Функционирование конвейера OpenGL

OpenGL является прослойкой между аппаратурой и пользовательским уровнем, что позволяет предоставлять единый интерфейс на разных платформах, используя возможности аппаратной поддержки.

Кроме того, OpenGL можно рассматривать как конечный автомат, состояние которого определяется множеством значений специальных переменных и значениями текущей нормали, цвета, координат

текстуры и других атрибутов и признаков. Вся эта информация будет использована при поступлении в графическую систему координат вершины для построения фигуры, в которую она входит. Смена состояний происходит с помощью команд, которые оформляются как вызовы функций.

1.4. Синтаксис команд

Определения команд GL находятся в файле gl.h, для включения которого нужно написать

```
#include <gl/gl.h>
```

Для работы с библиотекой GLU нужно аналогично включить файл glu.h. Версии этих библиотек, как правило, включаются в дистрибутивы систем программирования, например Microsoft Visual C++ или Borland C++ 5.02.

В отличие от стандартных библиотек, пакет GLUT нужно инсталлировать и подключать отдельно. Подробная информация о настройке сред программирования для работы с OpenGL дана в Приложении С.

Все команды (процедуры и функции) библиотеки GL начинаются с префикса gl, все константы – с префикса GL_. Соответствующие команды и константы библиотек GLU и GLUT аналогично имеют префиксы glu (GLU_) и glut (GLUT_)

Кроме того, в имена команд входят суффиксы, несущие информацию о числе и типе передаваемых параметров. В OpenGL полное имя команды имеет вид:

```
type glCommand_name[1 2 3 4][b s i f d ub us ui][v]
                   (type1 arg1,…,typeN argN)
```

Имя состоит из нескольких частей:

gl	имя библиотеки, в которой описана эта функция: для базовых функций OpenGL, функций из библиотек GL, GLU, GLUT, GLAUX это gl, glu, glut, aux соответственно.
Command_name	имя команды (процедуры или функции)

[1 2 3 4]	число аргументов команды
[b s i f d ub us ui]	тип аргумента: символ b – GLbyte (аналог char в C\C++), символ i – GLint (аналог int), символ f – GLfloat (аналог float) и так далее. Полный список типов и их описание можно посмотреть в файле gl.h
[v]	наличие этого символа показывает, что в качестве параметров функции используется указатель на массив значений

Символы в квадратных скобках в некоторых названиях не используются. Например, команда `glVertex2i()` описана в библиотеке GL, и использует в качестве параметров два целых числа, а команда `glColor3fv()` использует в качестве параметра указатель на массив из трех вещественных чисел.

Использования нескольких вариантов каждой команды можно частично избежать, применяя перегрузку функций языка C++. Но интерфейс OpenGL не рассчитан на конкретный язык программирования, и, следовательно, должен быть максимально универсален.

1.5. Пример приложения

Типичная программа, использующая OpenGL, начинается с определения окна, в котором будет происходить отображение. Затем создается контекст (клиент) OpenGL и ассоциируется с этим окном. Далее программист может свободно использовать команды и операции OpenGL API.

Ниже приведен текст небольшой программы, написанной с использованием библиотеки GLUT – своеобразный аналог классического примера “Hello, World!”.

Все, что делает эта программа – рисует в центре окна красный квадрат. Тем не менее, даже на этом простом примере можно понять принципы программирования с помощью OpenGL.

```
#include <stdlib.h>

/* подключаем библиотеку GLUT */
#include <gl/glut.h>

/* начальная ширина и высота окна */
```

```
GLint Width = 512, Height = 512;

/* размер куба */
const int CubeSize = 200;

/* эта функция управляет всем выводом на экран */
void Display(void)
{
    int left, right, top, bottom;

    left  = (Width - CubeSize) / 2;
    right = left + CubeSize;
    bottom = (Height - CubeSize) / 2;
    top = bottom + CubeSize;

    glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3ub(255,0,0);
    glBegin(GL_QUADS);
      glVertex2f(left,bottom);
      glVertex2f(left,top);
      glVertex2f(right,top);
      glVertex2f(right,bottom);
    glEnd();

    glFinish();
}

/* Функция вызывается при изменении размеров окна */
void Reshape(GLint w, GLint h)
{
    Width = w;
    Height = h;

    /* устанавливаем размеры области отображения */
    glViewport(0, 0, w, h);

    /* ортографическая проекция */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, w, 0, h, -1.0, 1.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
/* Функция обрабатывает сообщения от клавиатуры */
void
Keyboard( unsigned char key, int x, int y )
{
#define ESCAPE '\033'

    if( key == ESCAPE )
        exit(0);
}

/* Главный цикл приложения */
main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowSize(Width, Height);
    glutCreateWindow("Red square example");

    glutDisplayFunc(Display);
    glutReshapeFunc(Reshape);
    glutKeyboardFunc(Keyboard);

    glutMainLoop();
}
```

Несмотря на малый размер, это полностью завершенная программа, которая должна компилироваться и работать на любой системе, поддерживающей OpenGL и GLUT.

Библиотека GLUT поддерживает взаимодействие с пользователем с помощью так называемых функций с обратным вызовом (*callback function*). Если пользователь подвинул мышь, нажал на кнопку клавиатуры или изменил размеры окна, происходит событие и вызывается соответствующая функция пользователя – обработчик событий (функция с обратным вызовом).

Рассмотрим более подробно функцию `main` данного примера. Она состоит из трех частей – инициализации окна, в котором будет рисовать OpenGL, настройки функций с обратным вызовом и главного цикла обработки событий.

Инициализация окна состоит из настройки соответствующих буферов кадра, начального положения и размеров окна, а также заголовка окна.

Функция `glutInit(&argc, argv)` производит начальную инициализацию самой библиотеки GLUT.

Команда `glutInitDisplayMode(GLUT_RGB)` инициализирует буфер кадра и настраивает полноцветный (непалитровый) режим RGB.

`glutInitWindowSize(Width, Height)` используется для задания начальных размеров окна.

Наконец, `glutCreateWindow("Red square example")` задает заголовок окна и визуализирует само окно на экране.

Затем команды

```
glutDisplayFunc(Display);
glutReshapeFunc(Reshape);
glutKeyboardFunc(Keyboard);
```

регистрируют функции `Display()`, `Reshape()` и `Keyboard()` как функции, которые будут вызваны, соответственно, при перерисовке окна, изменении размеров окна, нажатии клавиши на клавиатуре.

Контроль всех событий и вызов нужных функций происходит внутри бесконечного цикла в функции `glutMainLoop()`

Заметим, что библиотека GLUT не входит в состав OpenGL, а является лишь переносимой прослойкой между OpenGL и оконной подсистемой, предоставляя минимальный интерфейс. OpenGL-приложение для конкретной платформы может быть написано с использованием специфических API (Win32, X Window и т.д.), которые как правило предоставляют более широкие возможности.

Более подробно работа с библиотекой GLUT описана в Приложении А.

Все вызовы команд OpenGL происходят в обработчиках событий. Более подробно они будут рассмотрены в следующих главах. Сейчас обратим внимание на функцию Display, в которой сосредоточен код, непосредственно отвечающий за рисование на экране.

Следующая последовательность команд из функции Display

```
glClearColor(0, 0, 0, 1);
glClear(GL_COLOR_BUFFER_BIT);

glColor3ub(255,0,0);
glBegin(GL_QUADS);
```

```
  glVertex2f(left,bottom);
  glVertex2f(left,top);
  glVertex2f(right,top);
  glVertex2f(right,bottom);
glEnd();
```

очищает окно и выводит на экран квадрат, задавая координаты четырех угловых вершин и цвет.

В приложении D.1 приведен еще один пример несложной программы, при нажатии кнопку мыши рисующей на экране разноцветные случайные прямоугольники.

Контрольные вопросы:

1. В чем, по вашему мнению, заключается необходимость создания стандартной графической библиотеки?
2. Кратко опишите архитектуру библиотек OpenGL и организацию конвейера.
3. В чем заключаются функции библиотек, подобных GLUT или GLX? Почему они формально не входят в OpenGL?
4. Назовите категории команд (функций) библиотеки.
5. Почему организацию OpenGL часто сравнивают с конечным автоматом?
6. Зачем нужны различные варианты команд OpenGL, отличающиеся только типами параметров?
7. Что можно сказать о количестве и типе параметров команды `glColor4ub()? glVertex3fv()?`

Глава 2

Рисование геометрических объектов

2.1. Процесс обновления изображения

Как правило, задачей программы, использующей OpenGL, является обработка трехмерной сцены и интерактивное отображение в буфере кадра. Сцена состоит из набора трехмерных объектов, источников света и виртуальной камеры, определяющей текущее положение наблюдателя.

Обычно приложение OpenGL в бесконечном цикле вызывает функцию обновления изображения в окне. В этой функции и сосредоточены вызовы основных команд OpenGL. Если используется библиотека GLUT, то это будет функция с обратным вызовом, зарегистрированная с помощью вызова `glutDisplayFunc()`. GLUT вызывает эту функцию, когда операционная система информирует приложение о том, что содержимое окна необходимо перерисовать (например, если окно было перекрыто другим). Создаваемое изображение может быть как статичным, так и анимированным, т.е. зависеть от каких-либо параметров, изменяющихся со временем. В этом случае лучше вызывать функцию обновления самостоятельно. Например, с помощью команды `glutPostRedisplay()`. За более подробной информацией можно обратиться к приложению А.

Приступим, наконец, к тому, чем занимается типичная функция обновления изображения. Как правило, она состоит из трех шагов:

1. очистка буферов OpenGL;
2. установка положения наблюдателя;
3. преобразование и рисование геометрических объектов.

Очистка буферов производится с помощью команды:

```
void glClearColor ( clampf r, clampf g, clampf b,
                    clampf a )
void glClear(bitfield buf)
```

Команда `glClearColor` устанавливает цвет, которым будет заполнен буфер кадра. Первые три параметра команды задают R,G и B компоненты цвета и должны принадлежать отрезку [0,1]. Четвертый параметр задает так называемую альфа компоненту (см. п. 6.1). Как правило, он равен 1. По умолчанию цвет – черный (0,0,0,1).

Команда `glClear` очищает буферы, а параметр *buf* определяет комбинацию констант, соответствующую буферам, которые нужно очистить (см. главу 6). Типичная программа вызывает команду

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

для очистки буферов цвета и глубины.

Установка положения наблюдателя и преобразования трехмерных объектов (поворот, сдвиг и т.д.) контролируются с помощью задания матриц преобразования. Преобразования объектов и настройка положения виртуальной камеры описаны в главе 3.

Сейчас сосредоточимся на том, как передать в OpenGL описания объектов, находящихся в сцене. Каждый объект является набором примитивов OpenGL.

2.2. Вершины и примитивы

Вершина является атомарным графическим примитивом OpenGL и определяет точку, конец отрезка, угол многоугольника и т.д. Все остальные примитивы формируются с помощью задания вершин, входящих в данный примитив. Например, отрезок определяется двумя вершинами, являющимися концами отрезка.

С каждой вершиной ассоциируются ее *атрибуты*. В число основных атрибутов входят положение вершины в пространстве, цвет вершины и вектор нормали.

2.2.1. Положение вершины в пространстве

Положение вершины определяются заданием ее координат в двух-, трех-, или четырехмерном пространстве (однородные координаты). Это реализуется с помощью нескольких вариантов команды `glVertex*`:

```
void glVertex[2 3 4][s i f d] (type coords)
void glVertex[2 3 4][s i f d]v (type *coords)
```

Каждая команда задает четыре координаты вершины: x, y, z, w. Команда `glVertex2*` получает значения x и y. Координата z в таком случае устанавливается по умолчанию равной 0, координата w – равной 1. `Vertex3*` получает координаты x, y, z и заносит в координату w значение 1. `Vertex4*` позволяет задать все четыре координаты.

Для ассоциации с вершинами цветов, нормалей и текстурных координат используются текущие значения соответствующих данных, что отвечает организации OpenGL как конечного автомата. Эти значения могут быть изменены в любой момент с помощью вызова соответствующих команд.

2.2.2. Цвет вершины

Для задания текущего цвета вершины используются команды :

```
void glColor[3 4][b s i f] (GLtype components)
void glColor[3 4][b s i f]v (GLtype components)
```

Первые три параметра задают R, G, B компоненты цвета, а последний параметр определяет коэффициент непрозрачности (так называемая альфа-компонента). Если в названии команды указан тип ‘f’ (float), то значения всех параметров должны принадлежать отрезку [0,1], при этом по умолчанию значение альфа-компоненты устанавливается равным 1.0, что соответствует полной непрозрачности. Тип ‘ub’ (unsigned byte) подразумевает, что значения должны лежать в отрезке [0,255].

Вершинам можно назначать различные цвета, и, если включен соответствующий режим, то будет проводиться линейная интерполяция цветов по поверхности примитива.

Для управления режимом интерполяции используется команда

```
void glShadeModel (GLenum  mode)
```

вызов которой с параметром **GL_SMOOTH** включает интерполяцию (установка по умолчанию), а с **GL_FLAT** – отключает.

2.2.3. Нормаль

Определить нормаль в вершине можно, используя команды

```
void glNormal3[b s i f d] (type coords)
void glNormal3[b s i f d]v (type coords)
```

Для правильного расчета освещения необходимо, чтобы вектор нормали имел единичную длину. Командой `glEnable(GL_NORMALIZE)`можно включить специальный режим, при котором задаваемые нормали будут нормироваться автоматически.

Режим автоматической нормализации должен быть включен, если приложение использует модельные преобразования растяжения/сжатия, так как в этом случае длина нормалей изменяется при умножении на модельно-видовую матрицу.

Однако применение этого режима уменьшает скорость работы механизма визуализации OpenGL, так как нормализация векторов имеет заметную вычислительную сложность (взятие квадратного корня и т.п.). Поэтому лучше сразу задавать единичные нормали.

Отметим, что команды

```
void glEnable (GLenum mode)
void glDisable (GLenum mode)
```

производят включение и отключение того или иного режима работы конвейера OpenGL. Эти команды применяются достаточно часто, и их возможные параметры будут рассматриваться в каждом конкретном случае.

2.3. Операторные скобки glBegin / glEnd

Мы рассмотрели задание атрибутов одной вершины. Однако, чтобы задать атрибуты графического примитива, одних координат вершин недостаточно. Эти вершины надо объединить в одно целое, определив необходимые свойства. Для этого в OpenGL используются так называемые операторные скобки, являющиеся вызовами специальных команд OpenGL Определение примитива или последовательности примитивов происходит между вызовами команд

```
void glBegin (GLenum mode);
void glEnd (void);
```

Параметр *mode* определяет тип примитива, который задается внутри и может принимать следующие значения:

GL_POINTS	каждая вершина задает координаты некоторой точки.
GL_LINES	каждая отдельная пара вершин определяет отрезок; если задано нечетное число вершин, то последняя вершина игнорируется.
GL_LINE_STRIP	каждая следующая вершина задает отрезок вместе с предыдущей.
GL_LINE_LOOP	отличие от предыдущего примитива только в том, что последний отрезок определяется последней и первой вершиной, образуя замкнутую ломаную.
GL_TRIANGLES	каждые отдельные три вершины определяют треугольник; если задано не кратное трем число вершин, то последние вершины игнорируются.
GL_TRIANGLE_STRIP	каждая следующая вершина задает треугольник вместе с двумя предыдущими.
GL_TRIANGLE_FAN	треугольники задаются первой вершиной и каждой следующей парой вершин (пары не пересекаются).
GL_QUADS	каждая отдельная четверка вершин определяет четырехугольник; если задано не кратное четырем число вершин, то последние вершины игнорируются.
GL_QUAD_STRIP	четырехугольник с номером n определяется вершинами с номерами 2n-1, 2n, 2n+2, 2n+1.
GL_POLYGON	последовательно задаются вершины выпуклого многоугольника.

Например, чтобы нарисовать треугольник с разными цветами в вершинах, достаточно написать:

```
GLfloat BlueCol[3] = {0,0,1};

glBegin(GL_TRIANGLES);
       glColor3f(1.0, 0.0, 0.0);  /* красный */
       glVertex3f(0.0, 0.0, 0.0);
       glColor3ub(0,255,0);       /* зеленый */
```

```
        glVertex3f(1.0, 0.0, 0.0);
        glColor3fv(BlueCol);       /* синий */
        glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

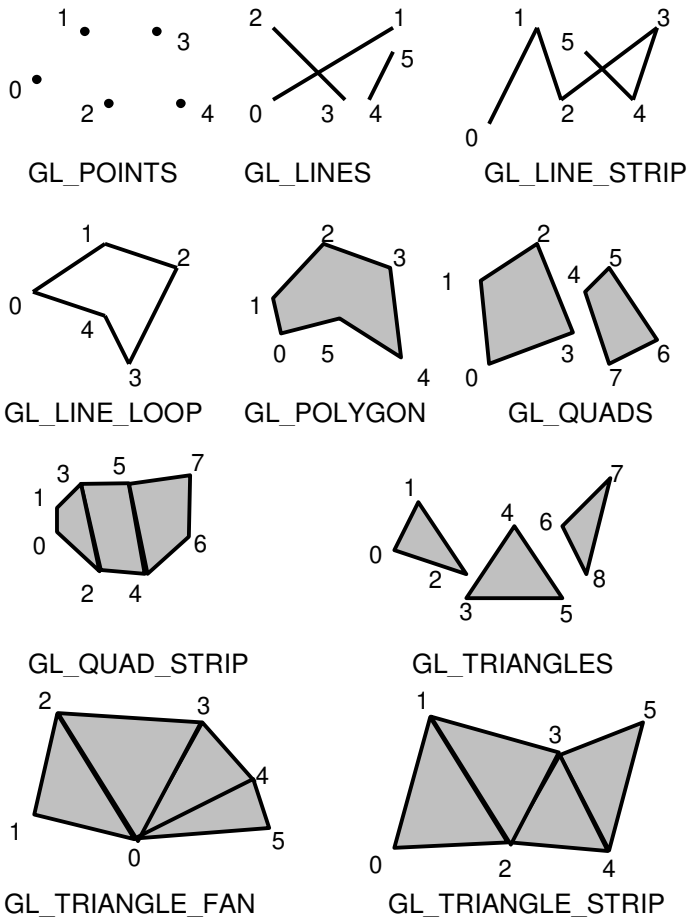


Рис. 3. Примитивы OpenGL

Как правило, разные типы примитивов имеют различную скорость визуализации на разных платформах. Для увеличения производительности предпочтительнее использовать примитивы, требующие меньшее количество информации для передачи на сервер, такие как **GL_TRIANGLE_STRIP**, **GL_QUAD_STRIP**, **GL_TRIAGLE_FAN.**

Кроме задания самих многоугольников, можно определить метод их отображения на экране.

Однако сначала надо определить понятие лицевых и обратных граней.

Под *гранью* понимается одна из сторон многоугольника, и по умолчанию лицевой считается та сторона, вершины которой обходятся против часовой стрелки. Направление обхода вершин лицевых граней можно изменить вызовом команды

```
void glFrontFace (GLenum mode)
```

со значением параметра *mode* равным **GL_CW** (clockwise), а вернуть значение по умолчанию можно, указав **GL_CCW** (counter-clockwise).

Чтобы изменить метод отображения многоугольника используется команда

```
void glPolygonMode (GLenum face, Glenum mode)
```

Параметр *mode* определяет, как будут отображаться многоугольники, а параметр *face* устанавливает тип многоугольников, к которым будет применяться эта команда и может принимать следующие значения:

GL_FRONT	для лицевых граней
GL_BACK	для обратных граней
GL_FRONT_AND_BACK	для всех граней

Параметр *mode* может быть равен:

GL_POINT	отображение только вершин многоугольников.
GL_LINE	многоугольники будут представляться набором отрезков.

GL_FILL многоугольники будут закрашиваться текущим цветом с учетом освещения, и этот режим установлен по умолчанию.

Также можно указывать, какой тип граней отображать на экране. Для этого сначала надо установить соответствующий режим вызовом команды **glEnable** (GL_CULL_FACE), а затем выбрать тип отображаемых граней с помощью команды

```
void glCullFace (GLenum mode)
```

Вызов с параметром **GL_FRONT** приводит к удалению из изображения всех лицевых граней, а с параметром **GL_BACK** – обратных (установка по умолчанию).

Кроме рассмотренных стандартных примитивов в библиотеках GLU и GLUT описаны более сложные фигуры, такие как сфера, цилиндр, диск (в GLU) и сфера, куб, конус, тор, тетраэдр, додекаэдр, икосаэдр, октаэдр и чайник (в GLUT). Автоматическое наложение текстуры предусмотрено только для фигур из библиотеки GLU (создание текстур в OpenGL будет рассматриваться в главе 5).

Например, чтобы нарисовать сферу или цилиндр, надо сначала создать объект специального типа GLUquadricObj с помощью команды

```
GLUquadricObj* gluNewQuadric(void);
```

а затем вызвать соответствующую команду:

```
void gluSphere (GLUquadricObj * qobj, GLdouble radius,
                GLint slices, GLint stacks)

void gluCylinder (GLUquadricObj * qobj,
                  GLdouble baseRadius,
                  GLdouble topRadius,
                  GLdouble height, GLint slices,
                  GLint stacks)
```

где параметр *slices* задает число разбиений вокруг оси z, а *stacks* – вдоль оси z.

Более подробную информацию об этих и других командах построения примитивов можно найти в приложении В.

2.4. Дисплейные списки

Если мы несколько раз обращаемся к одной и той же группе команд, то их можно объединить в так называемый дисплейный список (display list), и вызывать его при необходимости. Для того, чтобы создать новый дисплейный список, надо поместить все команды, которые должны в него войти, между следующими операторными скобками:

```
void glNewList (GLuint list, GLenum mode)
void glEndList ()
```

Для различения списков используются целые положительные числа, задаваемые при создании списка значением параметра *list*, а параметр *mode* определяет режим обработки команд, входящих в список:

GL_COMPILE	команды записываются в список без выполнения
GL_COMPILE_AND_EXECUTE	команды сначала выполняются, а затем записываются в список

После того, как список создан, его можно вызвать командой

```
void glCallList (GLuint list)
```

указав в параметре *list* идентификатор нужного списка. Чтобы вызвать сразу несколько списков, можно воспользоваться командой

```
void glCallLists (GLsizei n, GLenum type, const GLvoid
*lists)
```

вызывающей *n* списков с идентификаторами из массива *lists*, тип элементов которого указывается в параметре *type*. Это могут быть типы **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_INT**, **GL_UNSIGNED_INT** и некоторые другие. Для удаления списков используется команда

```
void glDeleteLists (GLint list, GLsizei range)
```

которая удаляет списки с идентификаторами ID из диапазона $list \leq ID \leq list+range-1$.

Пример:

```
glNewList(1, GL_COMPILE);
```

```
  glBegin(GL_TRIANGLES);
    glVertex3f(1.0f, 1.0f, 1.0f);
    glVertex3f(10.0f, 1.0f, 1.0f);
    glVertex3f(10.0f, 10.0f, 1.0f);
  glEnd();
glEndList()


   …
glCallList(1);
```

Дисплейные списки в оптимальном, скомпилированном виде хранятся в памяти сервера, что позволяет рисовать примитивы в такой форме максимально быстро. В то же время большие объемы данных занимают много памяти, что влечет, в свою очередь, падение производительности. Такие большие объемы (больше нескольких десятков тысяч примитивов) лучше рисовать с помощью массивов вершин.

2.5. Массивы вершин

Если вершин много, то чтобы не вызывать для каждой команду `glVertex*()`, удобно объединять вершины в массивы, используя команду

```
void glVertexPointer (GLint size, GLenum type,
                      GLsizei stride, void* ptr)
```

которая определяет способ хранения и координаты вершин. При этом *size* определяет число координат вершины (может быть равен 2, 3, 4), type определяет тип данных (может быть равен **GL_SHORT**, **GL_INT**, **GL_FLOAT**, **GL_DOUBLE**). Иногда удобно хранить в одном массиве другие атрибуты вершины, тогда параметр *stride* задает смещение от координат одной вершины до координат следующей; если *stride* равен нулю, это значит, что координаты расположены последовательно. В параметре *ptr* указывается адрес, где находятся данные.

Аналогично можно определить массив нормалей, цветов и некоторых других атрибутов вершины, используя команды

```
void glNormalPointer ( GLenum type, GLsizei stride,
                       void  *pointer )
void glColorPointer ( GLint size, GLenum type,
                      GLsizei stride, void *pointer )
```

Для того, чтобы эти массивы можно было использовать в дальнейшем, надо вызвать команду

```
void  glEnableClientState (GLenum array)
```

с параметрами **GL_VERTEX_ARRAY**, **GL_NORMAL_ARRAY**, **GL_COLOR_ARRAY** соответственно. После окончания работы с массивом желательно вызвать команду

```
void glDisableClientState (GLenum array)
```

с соответствующим значением параметра *array.*

Для отображения содержимого массивов используется команда

```
void glArrayElement (GLint index)
```

которая передает OpenGL атрибуты вершины, используя элементы массива с номером *index*. Это аналогично последовательному применению команд вида `glColor*(…)`, `glNormal*(…)`, `glVertex*(…)` с соответствующими параметрами. Однако вместо нее обычно вызывается команда

```
void glDrawArrays (GLenum mode, GLint first,
                   GLsizei count)
```

рисующая *count* примитивов, определяемых параметром *mode*, используя элементы из массивов с индексами от *first* до *first*+*count*-1. Это эквивалентно вызову последовательности команд `glArrayElement()` с соответствующими индексами.

В случае, если одна вершина входит в несколько примитивов, то вместо дублирования ее координат в массиве удобно использовать ее индекс.

Для этого надо вызвать команду

```
void glDrawElements (GLenum mode, GLsizei count,
                     GLenum type, void *indices)
```

где *indices* – это массив номеров вершин, которые надо использовать для построения примитивов, *type* определяет тип элементов этого

массива: **GL_UNSIGNED_BYTE**, **GL_UNSIGNED_SHORT**, **GL_UNSIGNED_INT**, а *count* задает их количество.

Важно отметить, что использование массивов вершин позволяет оптимизировать передачу данных на сервер OpenGL, и, как следствие, повысить скорость рисования трехмерной сцены. Такой метод определения примитивов является одним из самых быстрых и хорошо подходит для визуализации больших объемов данных.

Контрольные вопросы

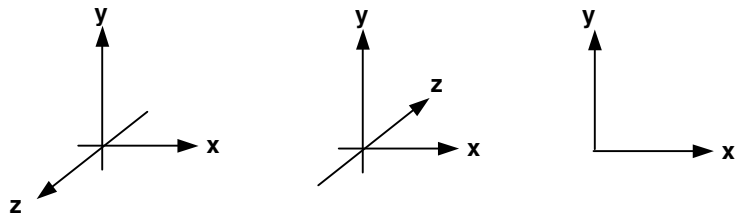
1. Что такое функция обратного вызова и как функции обратного вызова могут быть использованы для работы с OpenGL?
2. Для чего нужна функция обновления изображения и что она делает?
3. Что такое примитив в OpenGL?
4. Что такое атрибут? Перечислите известные вам атрибуты вершин в OpenGL.
5. Что в OpenGL является атомарным примитивом? Какие типы примитивов вы знаете?
6. Для чего в OpenGL используются команды glEnable/glDisable?
7. Что такое операторные скобки и для чего они используются в OpenGL?
8. Что такое дисплейные списки? Как определить список и как вызвать его отображение?
9. Поясните организацию работы с массивами вершин и их отличие от дисплейных списков.
10. Поясните работу команды glDrawElements()

Глава 3

Преобразования объектов

В OpenGL используются как основные три системы координат: левосторонняя, правосторонняя и оконная. Первые две системы являются трехмерными и отличаются друг от друга направлением оси z: в правосторонней она направлена на наблюдателя, в левосторонней – в глубину экрана. Ось x направлена вправо относительно наблюдателя, ось y – вверх.

Левосторонняя система используется для задания значений параметрам команды `gluPerspective()`, `glOrtho()`, которые будут рассмотрены в пункте 3.3. Правосторонняя система координат используется во всех остальных случаях. Отображение трехмерной информации происходит в двумерную *оконную* систему координат.



а) правосторонняя система б) левосторонняя система б) оконная система

Рис. 4 Системы координат в OpenGL

Строго говоря, OpenGL позволяет путем манипуляций с матрицами моделировать как правую, так и левую систему координат. Но на данном этапе лучше пойти простым путем и запомнить: основной системой координат OpenGL является правосторонняя система.

3.1. Работа с матрицами

Для задания различных преобразований объектов сцены в OpenGL используются операции над матрицами, при этом различают три типа матриц: модельно-видовая, матрица проекций и матрица текстуры. Все они имеют размер 4x4. Видовая матрица определяет преобразования

объекта в мировых координатах, такие как параллельный перенос, изменение масштаба и поворот. Матрица проекций определяет, как будут проецироваться трехмерные объекты на плоскость экрана (в оконные координаты), а матрица текстуры определяет наложение текстуры на объект.

Умножение координат на матрицы происходит в момент вызова соответствующей команды OpenGL, определяющей координату (как правило, это команда `glVertex*`)

Для того чтобы выбрать, какую матрицу надо изменить, используется команда:

```
void glMatrixMode (GLenum mode)
```

вызов которой со значением параметра *mode* равным **GL_MODELVIEW**, **GL_PROJECTION**, или **GL_TEXTURE** включает режим работы с модельно-видовой матрицей, матрицей проекций, или матрицей текстуры соответственно. Для вызова команд, задающих матрицы того или иного типа, необходимо сначала установить соответствующий режим.

Для определения элементов матрицы текущего типа вызывается команда

```
void glLoadMatrix[f d] (GLtype *m)
```

где *m* указывает на массив из 16 элементов типа float или double в соответствии с названием команды, при этом сначала в нем должен быть записан первый столбец матрицы, затем второй, третий и четвертый. Еще раз обратим внимание: в массиве *m* матрица записана *по столбцам*.

Команда

```
void  glLoadIdentity (void)
```

заменяет текущую матрицу на единичную.

Часто бывает необходимо сохранить содержимое текущей матрицы для дальнейшего использования, для чего применяются команды

```
void glPushMatrix (void)
void glPopMatrix (void)
```

Они записывают и восстанавливают текущую матрицу из стека, причем для каждого типа матриц стек свой. Для модельно-видовых матриц его глубина равна как минимум 32, для остальных – как минимум 2.

Для умножения текущей матрицы на другую матрицу используется команда

```
void glMultMatrix[f d] (GLtype *m)
```

где параметр *m* должен задавать матрицу размером 4x4. Если обозначить текущую матрицу за M, передаваемую матрицу за T, то в результате выполнения команды `glMultMatrix` текущей становится матрица M * T. Однако обычно для изменения матрицы того или иного типа удобно использовать специальные команды, которые по значениям своих параметров создают нужную матрицу и умножают ее на текущую.

В целом, для отображения трехмерных объектов сцены в окно приложения используется последовательность, показанная на рисунке.

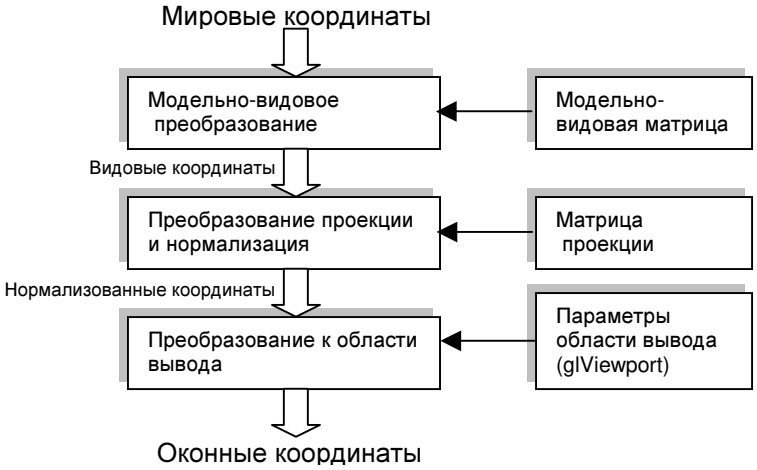


Рис. 5. Преобразования координат в OpenGL

Запомните: все преобразования объектов и камеры в OpenGL производятся с помощью умножения векторов координат на матрицы. Причем умножение происходит на *текущую матрицу* в момент определения координаты командой `glVertex*` и некоторыми другими.

3.2. Модельно-видовые преобразования

К модельно-видовым преобразованиям будем относить перенос, поворот и изменение масштаба вдоль координатных осей. Для проведения этих операций достаточно умножить на соответствующую матрицу каждую вершину объекта и получить измененные координаты этой вершины:

$$(x', y', z', 1)^T = M * (x, y, z, 1)^T$$

где M – матрица модельно-видового преобразования. Перспективное преобразование и проектирование производится аналогично. Сама матрица может быть создана с помощью следующих команд:

```
void glTranslate[f d] (GLtype x, GLtype y, GLtype z)
void glRotate[f d] (GLtype angle, GLtype x, GLtype y,
                    GLtype z)
void glScale[f d] (GLtype x, GLtype y, GLtype z)
```

`glTranlsate*()` производит перенос объекта, прибавляя к координатам его вершин значения своих параметров.

`glRotate*()` производит поворот объекта против часовой стрелки на угол *angle* (измеряется в градусах) вокруг вектора (x,y,z).

`glScale*()` производит масштабирование объекта (сжатие или растяжение) вдоль вектора (x,y,z), умножая соответствующие координаты его вершин на значения своих параметров.

Все эти преобразования изменяют текущую матрицу, а поэтому применяются к примитивам, которые определяются позже. В случае, если надо, например, повернуть один объект сцены, а другой оставить неподвижным, удобно сначала сохранить текущую видовую матрицу в стеке командой `glPushMatrix()`, затем вызвать `glRotate()` с нужными параметрами, описать примитивы, из которых состоит этот объект, а затем восстановить текущую матрицу командой `glPopMatrix()`.

Кроме изменения положения самого объекта, часто бывает необходимо изменить положение наблюдателя, что также приводит к изменению модельно-видовой матрицы. Это можно сделать с помощью команды

```
void gluLookAt
  (GLdouble eyex, GLdouble eyey, GLdouble eyez,
   GLdouble centerx, GLdouble centery, GLdouble centerz,
   GLdouble upx, GLdouble upy, GLdouble upz)
```

где точка (*eyex,eyey,eyez*) определяет точку наблюдения, (*centerx, centery, centerz*) задает центр сцены, который будет проектироваться в центр области вывода, а вектор (*upx,upy,upz*) задает положительное направление оси y, определяя поворот камеры. Если, например, камеру не надо поворачивать, то задается значение (0,1,0), а со значением (0,-1,0) сцена будет перевернута.

Строго говоря, эта команда совершает перенос и поворот объектов сцены, но в таком виде задавать параметры бывает удобнее. Следует отметить, что вызывать команду `gluLookAt()` имеет смысл *перед* определением преобразований объектов, когда модельно-видовая матрица равна единичной.

Запомните: В общем случае матричные преобразования в OpenGL нужно записывать в обратном порядке. Например, если вы хотите сначала повернуть объект, а затем передвинуть его, сначала вызовите команду `glTranslate()`, а только потом – `glRotate()`. Ну а после этого определяйте сам объект.

3.3. Проекции

В OpenGL существуют стандартные команды для задания ортографической (параллельной) и перспективной проекций. Первый тип проекции может быть задан командами

```
void glOrtho (GLdouble left, GLdouble right,
              GLdouble bottom, GLdouble top,
              GLdouble near, GLdouble far)

void gluOrtho2D (GLdouble left, GLdouble right, GLdouble
bottom, GLdouble top)
```

Первая команда создает матрицу проекции в усеченный объем видимости (параллелепипед видимости) в левосторонней системе координат. Параметры команды задают точки (*left, bottom, znear*) и (*right, top, zfar*), которые отвечают левому нижнему и правому верхнему углам окна вывода. Параметры *near* и *far* задают расстояние до ближней и дальней плоскостей отсечения по удалению от точки (0,0,0) и могут быть отрицательными.

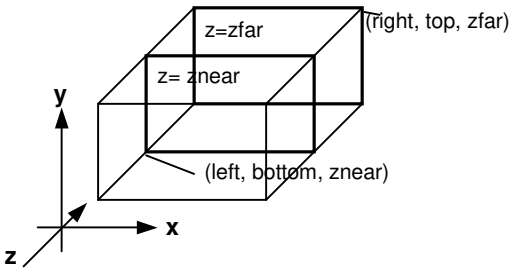


Рис. 6 Ортографическая проекция

Во второй команде, в отличие от первой, значения *near* и *far* устанавливаются равными –1 и 1 соответственно. Это удобно, если OpenGL используется для рисования двумерных объектов. В этом случае положение вершин можно задавать, используя команды `glVertex2*()`

Перспективная проекция определяется командой

```
void gluPerspective (GLdouble angley, GLdouble aspect,
                     GLdouble znear, GLdouble zfar)
```

которая задает усеченный конус видимости в левосторонней системе координат. Параметр *angley* определяет угол видимости в градусах по оси y и должен находиться в диапазоне от 0 до 180. Угол видимости вдоль оси x задается параметром *aspect*, который обычно задается как отношение сторон области вывода (как правило, размеров окна).

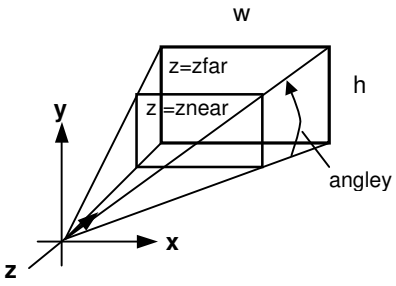


Рис. 7 Перспективная проекция

Параметры *zfar* и *znear* задают расстояние от наблюдателя до плоскостей отсечения по глубине и должны быть положительными. Чем больше отношение *zfar*/*znear*, тем хуже в буфере глубины будут различаться расположенные рядом поверхности, так как по умолчанию в него будет записываться ‘сжатая’ глубина в диапазоне от 0 до 1 (см. п. 3.4.).

Прежде чем задавать матрицы проекций, не забудьте включить режим работы с нужной матрицей командой `glMatrixMode(GL_PROJECTION)` и сбросить текущую, вызвав `glLoadIdentity()`.

Пример:

```
/* ортографическая проекция */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, 0, h, -1.0, 1.0);
```

3.4. Область вывода

После применения матрицы проекций на вход следующего преобразования подаются так называемые усеченные (clipped) координаты. Затем находятся нормализованные координаты вершин по формуле:

$(x_n, y_n, z_n)^T = (x_c/w_c, y_c/w_c, z_c/w_c)^T$

Область вывода представляет собой прямоугольник в оконной системе координат, размеры которого задаются командой:

```
void glViewPort (GLint x, GLint y, GLint width, GLint
height)
```

Значения всех параметров задаются в пикселах и определяют ширину и высоту области вывода с координатами левого нижнего угла (*x*,*y*) в оконной системе координат. Размеры оконной системы координат определяются текущими размерами окна приложения, точка (0,0) находится в левом нижнем углу окна.

Используя параметры команды `glViewPort()`, OpenGL вычисляет оконные координаты центра области вывода (o_x,o_y) по формулам o_x=x+width/2, o_y=y+height/2.

Пусть p_x=width, p_y=height, тогда можно найти оконные координаты каждой вершины:

$$(x_w, y_w, z_w)^T = ((p_x/2)\, x_n + o_x , (p_y/2)\, y_n + o_y , [(f-n)/2]\, z_n + (n+f)/2)^T$$

При этом целые положительные величины n и f задают минимальную и максимальную глубину точки в окне и по умолчанию равны 0 и 1 соответственно. Глубина каждой точки записывается в специальный буфер глубины (z-буфер), который используется для удаления невидимых линий и поверхностей. Установить значения n и f можно вызовом функции

```
void glDepthRange (GLclampd n, GLclampd f)
```

Команда `glViewPort()` обычно используется в функции, зарегистрированной с помощью команды `glutReshapeFunc()`, которая вызывается, если пользователь изменяет размеры окна приложения.

Контрольные вопросы

1. Какие системы координат используются в OpenGL?
2. Перечислите виды матричных преобразований в OpenGL. Каким образом происходят преобразования объектов в OpenGL?
3. Что такое матричный стек?
4. Перечислите способы изменения положения наблюдателя в OpenGL.
5. Какая последовательность вызовов команд glTranslate(), glRotate() и glScale() соответствует команде gluLookAt(0, 0, -10, 10, 0, 0, 0, -1, 0)?
6. Какие стандартные команды для задания проекций вы знаете?
7. Что такое видовые координаты? Нормализованные координаты?

Глава 4

Материалы и освещение

Для создания реалистичных изображений необходимо определить как свойства самого объекта, так и свойства среды, в которой он находится. Первая группа свойств включает в себя параметры материала, из которого сделан объект, способы нанесения текстуры на его поверхность, степень прозрачности объекта. Ко второй группе можно отнести количество и свойства источников света, уровень прозрачности среды, а также модель освещения. Все эти свойства можно задавать, вызывая соответствующие команды OpenGL.

4.1. Модель освещения

В OpenGL используется модель освещения, в соответствии с которой цвет точки определяется несколькими факторами: свойствами материала и текстуры, величиной нормали в этой точке, а также положением источника света и наблюдателя. Для корректного расчета освещенности в точке надо использовать единичные нормали, однако команды типа `glScale*()`, могут изменять длину нормалей. Чтобы это учитывать, используйте уже упоминавшийся в пункте 2.2.3 режим нормализации векторов нормалей, который включается вызовом команды `glEnable(GL_NORMALIZE)`.

Для задания глобальных параметров освещения используются команды

```
void glLightModel[i f] (GLenum pname, GLenum param)
void glLightModel[i f]v (GLenum pname,
                         const GLtype *params)
```

Аргумент *pname* определяет, какой параметр модели освещения будет настраиваться и может принимать следующие значения:

GL_LIGHT_MODEL_LOCAL_VIEWER параметр *param* должен быть булевым и задает положение наблюдателя. Если он равен **GL_FALSE**, то направление обзора считается параллельным

оси –z, вне зависимости от положения в видовых координатах. Если же он равен **GL_TRUE**, то наблюдатель находится в начале видовой системы координат. Это может улучшить качество освещения, но усложняет его расчет. Значение по умолчанию: **GL_FALSE**.

GL_LIGHT_MODEL_TWO_SIDE параметр *param* должен быть булевым и управляет режимом расчета освещенности, как для лицевых, так и для обратных граней. Если он равен **GL_FALSE**, то освещенность рассчитывается только для лицевых граней. Если же он равен **GL_TRUE**, расчет проводится и для обратных граней. Значение по умолчанию: **GL_FALSE**.

GL_LIGHT_MODEL_AMBIENT параметр *params* должен содержать четыре целых или вещественных числа, которые определяют цвет фонового освещения даже в случае отсутствия определенных источников света. Значение по умолчанию: (0.2, 0.2, 0.2,1.0).

4.2. Спецификация материалов

Для задания параметров текущего материала используются команды

```
void glMaterial[i f] (GLenum face, GLenum pname,
                      GLtype param)
void glMaterial[i f]v (GLenum face, GLenum pname,
                       GLtype *params)
```

С их помощью можно определить рассеянный, диффузный и зеркальный цвета материала, а также степень зеркального отражения и интенсивность излучения света, если объект должен светиться. Какой именно параметр будет определяться значением *param*, зависит от значения *pname*:

GL_AMBIENT параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют рассеянный цвет материала (цвет материала в тени). Значение по умолчанию: (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют диффузный цвет материала. Значение по умолчанию: (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют зеркальный цвет материала. Значение по умолчанию: (0.0, 0.0, 0.0, 1.0).

GL_SHININESS параметр *params* должен содержать одно целое или вещественное значение в диапазоне от 0 до 128, которое определяет степень зеркального отражения материала. Значение по умолчанию: 0.

GL_EMISSION параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют интенсивность излучаемого света материала. Значение по умолчанию: (0.0, 0.0, 0.0, 1.0).

GL_AMBIENT_AND_DIFFUSE эквивалентно двум вызовам команды `glMaterial*()` со значением pname **GL_AMBIENT** и **GL_DIFFUSE** и одинаковыми значениями *params*.

Из этого следует, что вызов команды `glMaterial[i f]()` возможен только для установки степени зеркального отражения материала (shininess). Команда `glMaterial[i f]v()` используется для задания остальных параметров.

Параметр *face* определяет тип граней, для которых задается этот материал и может принимать значения **GL_FRONT**, **GL_BACK** или **GL_FRONT_AND_BACK**.

Если в сцене материалы объектов различаются лишь одним параметром, рекомендуется сначала установить нужный режим, вызвав `glEnable()` с параметром **GL_COLOR_MATERIAL**, а затем использовать команду

`void` **`glColorMaterial`** `(GLenum` *`face`*`, GLenum` *`pname`*`)`

где параметр *face* имеет аналогичный смысл, а параметр *pname* может принимать все перечисленные значения. После этого значения выбранного с помощью *pname* свойства материала для конкретного объекта (или вершины) устанавливаются вызовом команды `glColor*()`, что позволяет избежать вызовов более ресурсоемкой команды `glMaterial*()` и повышает эффективность программы. Другие методы оптимизации приведены в п. 8.2.

Пример определения свойств материала:

```
float mat_dif[]={0.8,0.8,0.8};
float mat_amb[] = {0.2, 0.2, 0.2};
float mat_spec[] = {0.6, 0.6, 0.6};
float shininess = 0.7 * 128;
…
glMaterialfv (GL_FRONT_AND_BACK,GL_AMBIENT, mat_amb);
glMaterialfv (GL_FRONT_AND_BACK,GL_DIFFUSE, mat_dif);
glMaterialfv (GL_FRONT_AND_BACK,GL_SPECULAR, mat_spec);
glMaterialf  (GL_FRONT,GL_SHININESS, shininess);
```

4.3. Описание источников света

Определение свойств материала объекта имеет смысл, только если в сцене есть источники света. Иначе все объекты будут черными (или, строго говоря, иметь цвет, равный рассеянному цвету материала, умноженному на интенсивность глобального фонового освещения, см. команду glLightModel). Добавить в сцену источник света можно с помощью команд

```
void glLight[i f] (GLenum light, GLenum pname,
                   GLfloat param)
void glLight[i f] (GLenum light, GLenum pname,
                   GLfloat *params)
```

Параметр *light* однозначно определяет источник света. Он выбирается из набора специальных символических имен вида **GL_LIGHTi** , где i должно лежать в диапазоне от 0 до константы **GL_MAX_LIGHT**, которая обычно не превосходит восьми.

Параметры *pname* и *params* имеют смысл, аналогичный команде `glMaterial*()`. Рассмотрим значения параметра *pname*:

GL_SPOT_EXPONENT параметр *param* должен содержать целое или вещественное число от 0 до 128, задающее

распределение интенсивности света. Этот параметр описывает уровень сфокусированности источника света. Значение по умолчанию: 0 (рассеянный свет).

GL_SPOT_CUTOFF параметр *param* должен содержать целое или вещественное число между 0 и 90 или равное 180, которое определяет максимальный угол разброса света. Значение этого параметра есть половина угла в вершине конусовидного светового потока, создаваемого источником. Значение по умолчанию: 180 (рассеянный свет).

GL_AMBIENT параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет фонового освещения. Значение по умолчанию: (0.0, 0.0, 0.0, 1.0).

GL_DIFFUSE параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет диффузного освещения. Значение по умолчанию: (1.0, 1.0, 1.0, 1.0) для **GL_LIGHT0** и (0.0, 0.0, 0.0, 1.0) для остальных.

GL_SPECULAR параметр *params* должен содержать четыре целых или вещественных значения цветов RGBA, которые определяют цвет зеркального отражения. Значение по умолчанию: (1.0, 1.0, 1.0, 1.0) для **GL_LIGHT0** и (0.0, 0.0, 0.0, 1.0) для остальных.

GL_POSITION параметр *params* должен содержать четыре целых или вещественных числа, которые определяют положение источника света. Если значение компоненты w равно 0.0, то источник считается бесконечно удаленным и при расчете освещенности учитывается только направление на точку (x,y,z), в противном случае считается, что источник расположен в точке (x,y,z,w). В первом случае ослабления света при удалении от источника не

происходит, т.е. источник считается бесконечно удаленным. Значение по умолчанию: (0.0, 0.0, 1.0, 0.0).

GL_SPOT_DIRECTION параметр *params* должен содержать четыре целых или вещественных числа, которые определяют направление света. Значение по умолчанию: (0.0, 0.0, -1.0, 1.0). Эта характеристика источника имеет смысл, если значение **GL_SPOT_CUTOFF** отлично от 180 (которое, кстати, задано по умолчанию).

GL_CONSTANT_ATTENUATION,

GL_LINEAR_ATTENUATION,

GL_QUADRATIC_ATTENUATION

параметр *params* задает значение одного из трех коэффициентов, определяющих ослабление интенсивности света при удалении от источника. Допускаются только неотрицательные значения. Если источник не является направленным (см. **GL_POSITION**), то ослабление обратно пропорционально сумме:

$\mathbf{att}_{constant}+att_{linear}*d+ att_{quadratic} *d^2$,

где d – расстояние между источником света и освещаемой им вершиной, $\mathbf{att}_{constant}$, att_{linear} и $att_{quadratic}$ равны параметрам, заданным с помощью констант **GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION** и **GL_QUADRATIC_ATTENUATION** соответственно. По умолчанию эти параметры задаются тройкой (1, 0, 0), и фактически ослабления не происходит.

При изменении положения источника света следует учитывать следующий факт: в OpenGL источники света являются объектами, во многом такими же, как многоугольники и точки. На них распространяется основное правило обработки координат в OpenGL –

параметры, описывающее положение в пространстве, преобразуются текущей модельно-видовой матрицей в момент формирования объекта, т.е. в момент вызова соответствующих команд OpenGL. Таким образом, формируя источник света одновременно с объектом сцены или камерой, его можно привязать к этому объекту. Или, наоборот, сформировать стационарный источник света, который будет оставаться на месте, пока другие объекты перемещаются.

Общее правило такое:

Если положение источника света задается командой `glLight*()` перед определением положения виртуальной камеры (например, командой `glLookAt()`), то будет считаться, что координаты (0,0,0) источника находится в точке наблюдения и, следовательно, положение источника света определяется относительно положения наблюдателя.

Если положение устанавливается между определением положения камеры и преобразованиями модельно-видовой матрицы объекта, то оно фиксируется, т.е. в этом случае положение источника света задается в мировых координатах.

Для использования освещения сначала надо установить соответствующий режим вызовом команды `glEnable(GL_LIGHTNING)`, а затем включить нужный источник командой `glEnable(GL_LIGHTi)`.

Еще раз обратим внимание на то, что при выключенном освещении цвет вершины равен текущему цвету, который задается командами `glColor*()`. При включенном освещении цвет вершины вычисляется исходя из информации о материале, нормалях и источниках света.

При выключении освещения визуализация происходит быстрее, однако в таком случае приложение должно само рассчитывать цвета вершин.

Текст программы, демонстрирующей основные принципы определения материалов и источников света, приведен в приложении D.2

4.4. Создание эффекта тумана

В завершение рассмотрим одну интересную и часто используемую возможность OpenGL – создание эффекта тумана. Легкое затуманивание сцены создает реалистичный эффект, а частенько может

и скрыть некоторые артефакты, которые появляются, когда в сцене присутствуют отдаленные объекты.

Туман в OpenGL реализуется путем изменения цвета объектов в сцене в зависимости от их глубины, т.е. расстояния до точки наблюдения. Изменение цвета происходит либо для вершин примитивов, либо для каждого пикселя на этапе растеризации в зависимости от реализации OpenGL. Этим процессом можно частично управлять – см. раздел 6.4.

Для включения эффекта затуманивания необходимо вызвать команду **`glEnable(GL_FOG)`**.

Метод вычисления интенсивности тумана в вершине можно определить с помощью команд

```
void glFog[if] (enum pname, T param) ;
void glFog[if]v (enum pname, T params) ;
```

Аргумент *pname* может принимать следующие значения:

GL_FOG_MODE аргумент *param* определяет формулу, по которой будет вычисляться интенсивность тумана в точке.

В этом случае *param* может принимать значения

GL_EXP Интенсивность вычисляется по формуле f=exp(-d*z)

GL_EXP2 Интенсивность вычисляется по формуле $f=\exp(-(d*z)^2)$

GL_LINEAR Интенсивность вычисляется по формуле f=e-z/e-s

где z – расстояние от вершины, в которой вычисляется интенсивность тумана, до точки наблюдения.

Коэффициенты d,e,s задаются с помощью следующих значений аргумента *pname*

GL_FOG_DENSITY *param* определяет коээфициент d

GL_FOG_START *param* определяет коэффициент s

GL_FOG_END param определяет коэффициент e

Цвет тумана задается с помощью аргумента *pname*, равного

GL_FOG_COLOR в этом случае *params* – указатель на массив из 4-х компонент цвета.

Приведем пример использования этого эффекта:

```
GLfloat FogColor[4]={0.5,0.5,0.5,1};
glEnable(GL_FOG);
glFogi(GL_FOG_MODE,GL_LINEAR);
glFogf(GL_FOG_START,20.0);
glFogf(GL_FOG_END,100.0);
glFogfv(GL_FOG_COLOR,FogColor);
```

Контрольные вопросы

1. Поясните разницу между локальными и бесконечно удаленными источниками света.
2. Для чего служит команда glColorMaterial?
3. Как задать положение источника света таким образом, чтобы он всегда находился в точке положения наблюдателя?
4. Как задать фиксированное положение источника света? Можно ли задавать положение источника относительно локальных координат объекта?
5. Как задать конусный источник света?
6. Если в сцене включено освещение, но нет источников света, какой цвет будут иметь объекты?

Глава 5

Текстурирование

Под *текстурой* будем понимать некоторое изображение, которое надо определенным образом нанести на объект, например, для придания иллюзии рельефности поверхности.

Наложение текстуры на поверхность объектов сцены повышает ее реалистичность, однако при этом надо учитывать, что этот процесс требует вычислительных затрат, особенно если OpenGL не поддерживается аппаратно.

Для работы с текстурой следует выполнить следующую последовательность действий:

1. выбрать изображение и преобразовать его к нужному формату;
2. передать изображение в OpenGL;
3. определить, как текстура будет наноситься на объект и как она будет с ним взаимодействовать;
4. связать текстуру с объектом.

5.1. Подготовка текстуры

Для использования текстуры необходимо сначала загрузить в память нужное изображение и передать его OpenGL.

Считывание графических данных из файла и их преобразование можно проводить вручную. В приложении D приведен исходный текст функции для загрузки изображения из файла в формате BMP.

Можно также воспользоваться функцией, входящей в состав библиотеки GLAUX (для ее использования надо дополнительно подключить glaux.lib), которая сама проводит необходимые операции. Это функция

```
AUX_RGBImageRec* auxDIBImageLoad (const char *file)
```

где *file* – название файла с расширением *.bmp или *.dib. Функция возвращает указатель на область памяти, где хранятся преобразованные данные.

При создании образа текстуры в памяти следует учитывать следующие требования:

Во-первых, размеры текстуры, как по горизонтали, так и по вертикали должны представлять собой степени двойки. Это требование накладывается для компактного размещения текстуры в текстурной памяти и способствует ее эффективному использованию. Работать только с такими текстурами конечно неудобно, поэтому после загрузки их надо преобразовать. Изменение размеров текстуры можно провести с помощью команды

```
void gluScaleImage (GLenum format, GLint widthin,
                    GL heightin, GLenum typein,
                    const void *datain,
                    GLint widthout,
                    GLint heightout, GLenum typeout,
                    void *dataout)
```

В качестве значения параметра *format* обычно используется значение **GL_RGB** или **GL_RGBA**, определяющее формат хранения информации. Параметры *widthin*, *heightin*, *widhtout*, *heightout* определяют размеры входного и выходного изображений, а с помощью *typein* и *typeout* задается тип элементов массивов, расположенных по адресам *datain* и *dataout*. Как и обычно, это может быть тип **GL_UNSIGNED_BYTE, GL_SHORT, GL_INT** и так далее. Результат своей работы функция заносит в область памяти, на которую указывает параметр *dataout*.

Во-вторых, надо предусмотреть случай, когда объект после растеризации оказывается по размерам значительно меньше наносимой на него текстуры. Чем меньше объект, тем меньше должна быть наносимая на него текстура и поэтому вводится понятие *уровней детализации текстуры*. (mipmap) Каждый уровень детализации задает некоторое изображение, которое является, как правило, уменьшенной в два раза копией оригинала. Такой подход позволяет улучшить качество нанесения текстуры на объект. Например, для изображения размером $2^m x 2^n$ можно построить max(m,n)+1 уменьшенных изображений, соответствующих различным уровням детализации.

Эти два этапа создания образа текстуры во внутренней памяти OpenGL можно провести с помощью команды

```
void gluBuild2DMipmaps (GLenum target, GLint components,
                        GLint width, GLint height,
                        GLenum format, GLenum type,
                        const void *data)
```

где параметр *target* должен быть равен **GL_TEXTURE_2D.** Параметр *components* определяет количество цветовых компонент текстуры и может принимать следующие основные значения:

GL_LUMINANCE одна компонента – яркость. (текстура будет монохромной)

GL_RGB красный, синий, зеленый

GL_RGBA все компоненты.

Параметры *width*, *height*, *data* определяют размеры и расположение текстуры соответственно, а *format* и *type* имеют аналогичный смысл, что и в команде `gluScaleImage()`.

После выполнения этой команды текстура копируется во внутреннюю память OpenGL, и поэтому память, занимаемую исходным изображением, можно освободить.

В OpenGL допускается использование одномерных текстур, то есть размера 1xN, однако, это всегда надо указывать, задавая в качестве значения *target* константу **GL_TEXTURE_1D**. Полезность одномерных текстур сомнительна, поэтому не будем останавливаться на этом подробно.

При использовании в сцене нескольких текстур, в OpenGL применяется подход, напоминающий создание списков изображений (так называемые *текстурные объекты*). Сначала с помощью команды

```
void glGenTextures (GLsizei n, GLuint* textures)
```

надо создать *n* идентификаторов текстур, которые будут записаны в массив *textures*. Перед началом определения свойств очередной текстуры следует сделать ее текущей («привязать» текстуру), вызвав команду

```
void glBindTexture (GLenum target, GLuint texture)
```

где *target* может принимать значения **GL_TEXTURE_1D** или **GL_TEXTURE_2D**, а параметр *texture* должен быть равен идентификатору той текстуры, к которой будут относиться последующие команды. Для того, чтобы в процессе рисования сделать текущей текстуру с некоторым идентификатором, достаточно опять вызвать команду `glBindTexture()` с соответствующим значением *target* и *texture*. Таким образом, команда `glBindTexture()` включает режим создания текстуры с идентификатором *texture*, если такая текстура еще не создана, либо режим ее использования, то есть делает эту текстуру текущей.

Так как не всякая аппаратура может оперировать текстурами большого размера, целесообразно ограничить размеры текстуры до 256x256 или 512x512 пикселей. Отметим, что использование небольших текстур повышает эффективность программы.

5.2. Наложение текстуры на объекты

При наложении текстуры, как уже упоминалось, надо учитывать случай, когда размеры текстуры отличаются от оконных размеров объекта, на который она накладывается. При этом возможно как растяжение, так и сжатие изображения, и то, как будут проводиться эти преобразования, может серьезно повлиять на качество построенного изображения. Для определения положения точки на текстуре используется параметрическая система координат (s,t), причем значения s и t находятся в отрезке [0,1] (см. рисунок)

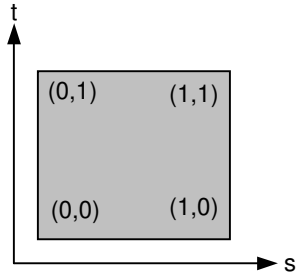


Рис. 8 Текстурные координаты

Для изменения различных параметров текстуры применяются команды:

```
void glTexParameter[i f] (GLenum target, GLenum pname,
                          GLenum param)
```

```
void glTexParameter[i f]v (GLenum target, GLenum pname,
                           Glenum* params)
```

При этом *target* может принимать значения **GL_TEXTURE_1D** или **GL_TEXTURE_2D**, *pname* определяет, какое свойство будем менять, а с помощью *param* или *params* устанавливается новое значение. Возможные значения *pname*:

GL_TEXTURE_MIN_FILTER параметр *param* определяет функцию, которая будет использоваться для сжатия текстуры. При значении **GL_NEAREST** будет использоваться один (ближайший), а при значении **GL_LINEAR** четыре ближайших элемента текстуры. Значение по умолчанию:**GL_LINEAR**.

GL_TEXTURE_MAG_FILTER параметр *param* определяет функцию, которая будет использоваться для увеличения (растяжения) текстуры. При значении **GL_NEAREST** будет использоваться один (ближайший), а при значении **GL_LINEAR** четыре ближайших элемента текстуры. Значение по умолчанию:**GL_LINEAR**.

GL_TEXTURE_WRAP_S параметр *param* устанавливает значение координаты s, если оно не входит в отрезок [0,1]. При значении **GL_ REPEAT** целая часть s отбрасывается, и в результате изображение размножается по поверхности. При значении **GL_CLAMP** используются краевые значения: 0 или 1, что удобно использовать, если на объект накладывается один образ. Значение по умолчанию: **GL_REPEAT**.

GL_TEXTURE_WRAP_T аналогично предыдущему значению, только для координаты t.

Использование режима **GL_NEAREST** повышает скорость наложения текстуры, однако при этом снижается качество, так как в отличие от **GL_LINEAR** интерполяция не производится.

Для того чтобы определить, как текстура будет взаимодействовать с материалом, из которого сделан объект, используются команды

```
void glTexEnv[i f]  (GLenum target, GLenum pname,
                     GLtype param)
void glTexEnv[i f]v (GLenum target, GLenum pname,
                     GLtype *params)
```

Параметр *target* должен быть равен **GL_TEXTURE_ENV**, а в качестве *pname* рассмотрим только одно значение **GL_TEXTURE_ENV_MODE**, которое наиболее часто применяется.

Наиболее часто используемые значения параметра *param*:

GL_MODULATE — конечный цвет находится как произведение цвета точки на поверхности и цвета соответствующей ей точки на текстуре.

GL_REPLACE — в качестве конечного цвета используется цвет точки на текстуре.

Следующая программа демонстрирует общий подход к созданию текстур:

```
/* нужное нам количество текстур */
#define NUM_TEXTURES 10
/* идентификаторы текстур */
int TextureIDs[NUM_TEXTURES];
/* образ текстуры */
AUX_RGBImageRec *pImage;

…
/* 1) получаем идентификаторы текстур */
glGenTextures(NUM_TEXTURES,TextureIDs);
/* 2) выбираем текстуру для модификации параметров */
glBindTexture(TextureIDs[i]); /* 0<=i<NUM_TEXTURES*/
/* 3) загружаем текстуру. Размеры текстуры – степень 2 */
pImage=dibImageLoad("texture.bmp");
if (Texture!=NULL)
{
  /* 4) передаем текстуру OpenGL и задаем параметры*/
  /* выравнивание по байту */
  glPixelStorei(GL_UNPACK_ALIGNMENT,1);
  gluBuildMipmaps(GL_TEXTURE_2D,GL_RGB, pImage->sizeX,
    pImage->sizeY, GL_RGB, GL_UNSIGNED_BYTE,
    pImage->data);
  glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
    (float)GL_LINEAR);
```

```
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    (float)GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    (float)GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    (float)GL_REPEAT);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
    (float)GL_REPLACE);
  /* 5) удаляем исходное изображение.*/
  free(Texture);
}else Error();
```

5.3. Текстурные координаты

Перед нанесением текстуры на объект необходимо установить соответствие между точками на поверхности объекта и на самой текстуре. Задавать это соответствие можно двумя методами: отдельно для каждой вершины или сразу для всех вершин, задав параметры специальной функции отображения.

Первый метод реализуется с помощью команд

```
void glTexCoord[1 2 3 4][s i f d]  (type coord)
void glTexCoord[1 2 3 4][s i f d]v (type *coord)
```

Чаще всего используется команды вида **glTexCoord2*(type s, type t)**, задающие текущие координаты текстуры. Понятие текущих координат текстуры аналогично понятиям текущего цвета и текущей нормали, и является атрибутом вершины. Однако даже для куба нахождение соответствующих координат текстуры является довольно трудоемким занятием, поэтому в библиотеке GLU помимо команд, проводящих построение таких примитивов, как сфера, цилиндр и диск, предусмотрено также наложение на них текстур. Для этого достаточно вызвать команду

```
void gluQuadricTexture (GLUquadricObj* quadObject,
GLboolean textureCoords)
```

с параметром *textureCoords* равным **GL_TRUE**, и тогда текущая текстура будет автоматически накладываться на примитив.

Второй метод реализуется с помощью команд

```
void glTexGen[i f d]  (GLenum coord, GLenum pname,
                       GLtype param)
```

```
void glTexGen[i f d]v (GLenum coord, GLenum pname,
                       const GLtype *params)
```

Параметр *coord* определяет, для какой координаты задается формула, и может принимать значение **GL_S,GL_T**; *pname* может быть равен одному из следующих значений:

GL_TEXTURE_GEN_MODE определяет функцию для наложения текстуры.

В этом случае аргумент *param* принимает значения:

GL_OBJECT_LINEAR значение соответствующей текстурной координаты определяется расстоянием до плоскости, задаваемой с помощью значения pname **GL_OBJECT_PLANE** (см. ниже). Формула выглядит следующим образом: g=x*xp+y*yp+z*zp+w*wp, где g– соответвующая текстурная координата (s или p), x, y, z, w – координаты соответствующей точки. xp, yp, zp, wp – коэффициенты уравнения плоскости. В формуле используются координаты объекта.

GL_EYE_LINEAR аналогично предыдущему значению, только в формуле используются видовые координаты. Т.е. координаты текстуры объекта в этом случае зависят от положения этого объекта.

GL_SPHERE_MAP позволяет эмулировать отражение от поверхности объекта. Текстура как бы "оборачивается" вокруг объекта. Для данного метода используются видовые координаты и необходимо задание нормалей.

GL_OBJECT_PLANE позволяет задать плоскость, расстояние до которой будет использоваться при генерации координат, если установлен режим **GL_OBJECT_LINEAR**. В этом случае параметр *params* является указателем на массив из четырех коэффициентов уравнения плоскости.

GL_EYE_PLANE аналогично предыдущему значению. Позволяет задать плоскость для режима **GL_EYE_LINEAR**

Для установки автоматического режима задания текстурных координат необходимо вызвать команду `glEnable` с параметром **GL_TEXTURE_GEN_S** или **GL_TEXTURE_GEN_P**.

Программа, использующая наложение текстуры и анимацию, приведена в приложении D.3

Контрольные вопросы

1. Что такое текстура и для чего используются текстуры?
2. Что такое текстурные координаты и как задать их для объекта?
3. Какой метод взаимодействия с материалом (GL_MODULATE, GL_REPLACE) нужно использовать, если текстура представляет собой картину, висящую на стене?
4. Перечислите известные вам методы генерации текстурных координат в OpenGL.
5. Для чего используются уровни детализации текстуры (mip-mapping)?
6. Что такое режимы фильтрации текстуры и как задать их в OpenGL?

Глава 6

Операции с пикселями

После проведения всех операций по преобразованию координат вершин, вычисления цвета и т.п., OpenGL переходит к этапу *растеризации*, на котором происходит растеризация всех примитивов, наложение текстуры, наложение эффекта тумана. Для каждого примитива результатом этого процесса является занимаемая им в буфере кадра область, каждому пикселю этой области приписывается цвет и значение глубины.

OpenGL использует эту информацию, чтобы записать обновленные данные в буфер кадра. Для этого OpenGL имеет не только отдельный конвейер обработки пикселей, но и несколько дополнительных буферов различного назначения. Это позволяет программисту гибко контролировать процесс визуализации на самом низком уровне.

Графическая библиотека OpenGL поддерживает работу со следующими буферами:

- несколько буферов цвета
- буфер глубины
- буфер-накопитель (аккумулятор)
- буфер маски

Группа буферов цвета включает буфер кадра, но таких буферов может быть несколько. При использовании двойной буферизации говорят о рабочем (front) и фоновом (back) буферах. Как правило, в фоновом буфере программа создает изображение, которое затем разом копируется в рабочий буфер. На экране может появиться информация только из буферов цвета.

Буфер глубины используется для удаления невидимых поверхностей и прямая работа с ним требуется крайне редко.

Буфер-накопитель можно применять для различных операций. Более подробно работа с ним описана в разделе 6.2.

Буфер маски используется для формирования пиксельных масок (трафаретов), служащих для вырезания из общего массива тех пикселей, которые следует вывести на экран. Буфер маски и работа с ним более подробно рассмотрены в разделах 6.3, 7.2 и 7.3.

6.1. Смешивание изображений. Прозрачность

Разнообразные прозрачные объекты – стекла, прозрачная посуда и т.д. часто встречаются в реальности, поэтому важно уметь создавать такие объекты в интерактивной графике. OpenGL предоставляет программисту механизм работы с полупрозрачными объектами, который и будет кратко описан в этом разделе.

Прозрачность реализуется с помощью специального режима смешения цветов (blending). Алгоритм смешения комбинирует цвета так называемых входящих пикселей (т.е. «кандидатов» на помещение в буфер кадра) с цветами соответствующих пикселей, уже хранящихся в буфере. Для смешения используется четвертая компонента цвета – альфа-компонента, поэтому этот режим называют еще альфа-смешиванием. Программа может управлять интенсивностью альфа-компоненты точно так же, как и интенсивностью основных цветов, т.е. задавать значение интенсивности для каждого пикселя или каждой вершины примитива.

Режим включается с помощью команды `glEnable(GL_BLEND)`.

Определить параметры смешения можно с помощью команды:

`void` **`glBlendFunc`**`(enum` *`src`*`,enum` *`dst`*`)`

Параметр *src* определяет, как получить коэффициент k_1 исходного цвета пикселя, а *dst* задает способ получения коэффициента k_2 для цвета в буфере кадра. Для получения результирующего цвета используется следующая формула: $res=c_{src}*k_1+c_{dst}*k_2$, где c_{src} – цвет исходного пикселя, c_{dst} – цвет пикселя в буфере кадра (res, k_1, k_1, $c_{src,}$ c_{dst} – четырехкомпонентные RGBA-векторы).

Приведем наиболее часто используемые значения агрументов `src` и `dst`.

GL_SRC_ALPHA	$k=(A_s,A_s,A_s,A_s)$
GL_SRC_ONE_MINUS_ALPHA	$k=(1,1,1,1)-(A_s,A_s,A_s,A_s)$
GL_DST_COLOR	$k=(R_d,G_d,B_d)$

GL_ONE_MINUS_DST_COLOR	k=(1,1,1,1)- (R_d,G_d,B_d,A_d)
GL_DST_ALPHA	k=(A_d,A_d,A_d,A_d)
GL_DST_ONE_MINUS_ALPHA	k=(1,1,1,1)-(A_d,A_d,A_d,A_d)
GL_SRC_COLOR	k=(R_s,G_s,B_s)
GL_ONE_MINUS_SRC_COLOR	k=(1,1,1,1)- (R_s,G_s,B_s,A_s)

Пример:

Предположим, мы хотим реализовать вывод прозрачных объектов. Коэффициент прозрачности задается альфа-компонентой цвета. Пусть 1 – непрозрачный объект; 0 – абсолютно прозрачный, т.е. невидимый. Для реализации служит следующий код:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_SRC_ONE_MINUS_ALPHA);
```

Например, полупрозрачный треугольник можно задать следующим образом:

```
glColor3f(1.0, 0.0, 0.0, 0.5);
glBegin(GL_TRIANGLES);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

Если в сцене есть несколько прозрачных объектов, которые могут перекрывать друг друга, корректный вывод можно гарантировать только в случае выполнения следующих условий:

- Все прозрачные объекты выводятся после непрозрачных.
- При выводе объекты с прозрачностью должны быть упорядочены по уменьшению глубины, т.е. выводиться, начиная с наиболее отдаленных от наблюдателя.

В OpenGL команды обрабатываются в порядке их поступления, поэтому для реализации перечисленных требований достаточно расставить в соответствующем порядке вызовы команд glVertex*(), но и это в общем случае нетривиально.

6.2. Буфер-накопитель

Буфер-накопитель (*accumulation buffer*) – это один из дополнительных буферов OpenGL. В нем можно сохранять визуализированное изображение, применяя при этом попиксельно специальные операции. Буфер-накопитель широко используется для создания различных спецэффектов.

Изображение берется из буфера, выбранного на чтение командой

```
void glReadBuffer(enum buf)
```

Аргумент *buf* определяет буфер для чтения. Значения *buf*, равные **GL_BACK**, **GL_FRONT**, определяют соответствующие буферы цвета для чтения. **GL_BACK** задает в качестве источника пикселей внеэкранный буфер; **GL_FRONT** – текущее содержимое окна вывода. Команда имеет значение, если используется дублирующая буферизация. В противном случае используется только один буфер, соответствующий окну вывода (строго говоря, OpenGL имеет набор дополнительных буферов, используемых, в частности, для работы со стереоизображениями, но здесь мы их рассматривать не будем).

Буфер-накопитель является дополнительным буфером цвета. Он не используется непосредственно для вывода образов, но они добавляются в него после вывода в один из буферов цвета. Применяя различные операции, описанные ниже, можно понемногу «накапливать» изображение в буфере.

Затем полученное изображение переносится из буфера-накопителя в один из буферов цвета, выбранный на запись командой

```
void glDrawBuffer(enum buf)
```

Значение *buf* аналогично значению соответствующего аргумента в команде **glReadBuffer**.

Все операции с буфером-накопителем контролируются командой

```
void glAccum(enum op,GLfloat value)
```

Аргумент *op* задает операцию над пикселями и может принимать следующие значения:

GL_LOAD	Пиксель берется из буфера, выбранного на чтение, его значение умножается на *value* и заносится в буфер-накопитель.
GL_ACCUM	Аналогично предыдущему, но полученное после умножения значение складывается с уже имеющимся в буфере.
GL_MULT	Эта операция умножает значение каждого пикселя в буфере накопления на *value* .
GL_ADD	Аналогично предыдущему, только вместо умножения используется сложение.
GL_RETURN	Изображение переносится из буфера накопителя в буфер, выбранный для записи. Перед этим значение каждого пикселя умножается на *value.*

Следует отметить, что для использования буфера-накопителя нет необходимости вызывать какие-либо команды `glEnable`. Достаточно инициализировать только сам буфер.

Пример использования буфера-накопителя для устранения погрешностей растеризации (ступенчатости) приведен в разделе 7.1

6.3. Буфер маски

При выводе пикселей в буфер кадра иногда возникает необходимость выводить не все пиксели, а только некоторое подмножество, т.е. наложить трафарет (маску) на изображение. Для этого OpenGL предоставляет так называемый буфер маски (stencil buffer). Кроме наложения маски, этот буфер предоставляет еще несколько интересных возможностей.

Прежде чем поместить пиксель в буфер кадра, механизм визуализации OpenGL позволяет выполнить сравнение (тест) между заданным значением и значением в буфере маски. Если тест проходит, пиксель рисуется в буфере кадра.

Механизм сравнения весьма гибок и контролируется следующими командами:

```
void glStencilFunc (enum func, int ref, uint mask)
void glStencilOp (enum sfail, enum dpfail, enum dppass)
```

Аргумент *ref* команды `glStencilFunc` задает значение для сравнения. Он должен принимать значение от 0 до 2^s –1. s – число бит на точку в буфере маски.

С помощью аргумента *func* задается функция сравнения. Он может принимать следующие значения:

GL_NEVER тест никогда не проходит, т.е. всегда возвращает false

GL_ALWAYS тест проходит всегда.

GL_LESS, GL_LEQUAL, GL_EQUAL,

GL_GEQUAL, GL_GREATE, GL_NOTEQUAL тест проходит в случае, если *ref* соответственно меньше значения в буфере маски, меньше либо равен, равен, больше, больше либо равен, или не равен.

Аргумент *mask* задает маску для значений. Т.е. в итоге для этого теста получаем следующую формулу: ((*ref* AND *mask*) *op* (svalue AND *mask*))

Команда `glStencilOp` предназначена для определения действий над пикселем буфера маски в случае положительного или отрицательного результата теста.

Аргумент *sfail* задает действие в случае отрицательного результата теста, и может принимать следующие значения:

GL_KEEP, GL_ZERO, GL_REPLACE,

GL_INCR, GL_DECR, GL_INVERT соответственно сохраняет значение в буфере маски, обнуляет его, заменяет на заданное значение (*ref*), увеличивает, уменьшает или побитово инвертирует.

Аргументы *dpfail* определяют действия в случае отрицательного результата теста на глубину в z-буфере, а *dppass* задает действие в случае положительного результата этого теста. Аргументы принимают те же значения, что и аргумент *sfail*. По умолчанию все три параметра установлены на **GL_KEEP**.

Для включения маскирования необходимо выполнить команду `glEnable(GL_STENCIL_TEST);`

Буфер маски используется при создании таких спецэффектов, как падающие тени, отражения, плавные переходы из одной картинки в другую и пр.

Пример использования буфера маски при построении теней и отражений приведен в разделах 7.3 и 7.2.

6.4. Управление растеризацией

Способ выполнения растеризации примитивов можно частично регулировать командой `glHint (target, mode)`, где *target* – вид контролируемых действий, принимает одно из следующих значений

GL_FOG_HINT – точность вычислений при наложении тумана. Вычисления могут выполняться по пикселям (наибольшая точность) или только в вершинах. Если реализация OpenGL не поддерживает попиксельного вычисления, то выполняется только вычисление по вершинам.

GL_LINE_SMOOTH_HINT – управление качеством прямых. При значении mode, равным **GL_NICEST**, уменьшается ступенчатость прямых за счет большего числа пикселей в прямых.

GL_PERSPECTIVE_CORRECTION_HINT – точность интерполяции координат при вычислении цветов и наложении текстуры. Если реализация OpenGL не поддерживает режим **GL_NICEST**, то осуществляется линейная интерполяция координат.

GL_POINT_SMOOTH_HINT – управление качеством точек. При значении параметра mode равным **GL_NICEST** точки рисуются как окружности.

GL_POLYGON_SMOOTH_HINT – управление качеством вывода сторон многоугольника.

Параметра *mode* интерпретируется следующим образом:

GL_FASTEST - используется наиболее быстрый алгоритм.

GL_NICEST - используется алгоритм, обеспечивающий лучшее качество.

GL_DONT_CARE - выбор алгоритма зависит от реализации.

Важно заметить, что командой `glHint()` программист может только определить свои пожелания относительно того или иного аспекта растеризации примитивов. Конкретная реализация OpenGL вправе игнорировать данные установки.

Обратите внимание, что `glHint()` нельзя вызывать между операторными скобками `glBegin()/glEnd()`.

Контрольные вопросы

1. Какие буферы изображений используются в OpenGL и для чего?
2. Для чего используется команда glBlendFunc?
3. Почему для корректного вывода прозрачных объектов требуется соблюдение условий упорядоченного вывода примитивов с прозрачностью?
4. Для чего используется буфер-накопитель? Приведите пример работы с ним.
5. Как в OpenGL можно наложить маску на результирующее изображение?
6. Объясните, для чего применятся команда glHint().
7. Каков эффект выполнения команды `glHint(`GL_FOG_HINT, GL_DONT_CARE)?

Глава 7

Приемы работы с OpenGL

В этой главе мы рассмотрим, как с помощью OpenGL создавать некоторые интересные визуальные эффекты, непосредственная поддержка которых отсутствует в стандарте библиотеки.

7.1. Устранение ступенчатости

Начнем с задачи устранения ступенчатости (*antialiasing*). Эффект ступенчатости (*aliasing*) возникает в результате погрешностей растеризации примитивов в буфере кадра из-за конечного (и, как, правило, небольшого) разрешения буфера. Есть несколько подходов к решению данной проблемы. Например, можно применять фильтрацию полученного изображения. Также этот эффект можно устранять на этапе растеризации, сглаживая образ каждого примитива. Здесь мы рассмотрим прием, позволяющий устранять подобные артефакты для всей сцены целиком.

Для каждого кадра необходимо нарисовать сцену несколько раз, на каждом проходе немного смещая камеру относительно начального положения. Положения камер, например, могут образовывать окружность. Если сдвиг камеры относительно мал, то погрешности дискретизации проявятся по-разному, и, усредняя полученные изображения, мы получим сглаженное изображение.

Проще всего сдвигать положение наблюдателя, но перед этим нужно вычислить размер сдвига так, чтобы приведенное к координатам экрана значение не превышало, скажем, половины размера пикселя.

Все полученные изображения сохраняем в буфере-накопителе с коэффициентом 1/n, где n – число проходов для каждого кадра. Чем больше таких проходов – тем ниже производительность, но лучше результат.

```
for(i=0;i<samples_count;++i)
/* обычно samples_count лежит в пределах от 5 до 10 */
{
  ShiftCamera(i); /* сдвигаем камеру */
```

```
  RenderScene();
  if (i==0)
   /* на первой итерации загружаем изображение */
   glAccum(GL_LOAD,1/(float)samples_count);
  else
   /* добавляем к уже существующему */
   glAccum(GL_ACCUM,1/(float)samples_count);
}
/* Пишем то, что получилось, назад в исходный буфер */
glAccum(GL_RETURN,1.0);
```

Следует отметить, что устранение ступенчатости сразу для всей сцены, как правило, связано с серьезным падением производительности визуализации, так как вся сцена рисуется несколько раз. Современные ускорители обычно аппаратно реализуют другие методы, основанные на так называемом ресамплинге изображений.

7.2. Построение теней

В OpenGL нет встроенной поддержки построения теней на уровне базовых команд. В значительной степени это объясняется тем, что существует множество алгоритмов их построения, которые могут быть реализованы через функции OpenGL. Присутствие теней сильно влияет на реалистичность трехмерного изображения, поэтому рассмотрим один из подходов к их построению.

Большинство алгоритмов, предназначенных для построения теней, используют модифицированные принципы перспективной проекции. Здесь рассматривается один из самых простых методов. С его помощью можно получать тени, отбрасываемые трехмерным объектом на плоскость.

Общий подход таков: для всех точек объекта находится их проекция параллельно вектору, соединяющему данную точку и точку, в которой находится источник света, на некую заданную плоскость. Тем самым получаем новый объект, целиком лежащий в заданной плоскости. Этот объект и является тенью исходного.

Рассмотрим математические основы данного метода.

Пусть:

P – точка в трехмерном пространстве, которая отбрасывает тень.

L – положение источника света, который освещает данную точку.

S=a(**L**-**P**)-**P** - точка, в которую отбрасывает тень точка P, где a – параметр.

Предположим, что тень падает на плоскость *z=0.* В этом случае $a=z_p/(z_l-z_p)$. Следовательно,

$x_s = (x_p z_l - z_l z_p) / (z_l - z_p)$,

$y_s = (y_p z_l - y_l z_p) / (z_l - z_p)$,

$z_s = 0$

Введем однородные координаты:

$x_s=x_s'/w_s'$

$y_s=y_s'/w_s'$

$z_s=0$

$w_s'=z_l-z_p$

Отсюда координаты S могут быть получены с использованием умножения матриц следующим образом:

$$[x_s' \quad y_s' \quad 0 \quad w_s'] = [x_s \quad y_s \quad z_s \quad 1] \begin{bmatrix} z_l & 0 & 0 & 0 \\ 0 & z_l & 0 & 0 \\ -x_l & -y_l & 0 & -1 \\ 0 & 0 & 0 & z_l \end{bmatrix}$$

Для того, чтобы алгоритм мог рассчитывать тень, падающую на произвольную плоскость, рассмотрим произвольную точку на линии между **S** и **P**, представленную в однородных координатах:

a**P**+b**L,** где a и b – скалярные параметры.

Следующая матрица задает плоскость через координаты ее нормали:

$$G = \begin{bmatrix} x_n \\ y_n \\ z_n \\ d \end{bmatrix}$$

Точка, в которой луч, проведенный от источника света через данную точку **P,** пересекает плоскость G, определяется параметрами a и b, удовлетворяющими следующему уравнению:

(a**P**+b**L**)**G** = 0

Отсюда получаем: a(**PG**) + b(**LG**) = 0. Этому уравнению удовлетворяют

a = (**LG**), b = -(**PG**)

Следовательно, координаты искомой точки **S** = (**LG**)**P**-(**PG**)**L.** Пользуясь ассоциативностью матричного произведения, получим

S = **P**[(**LG**)**I** - **GL**], где **I –** единичная матрица.

Матрица (**LG**)**I** - **GL** используется для получения теней на произвольной плоскости.

Рассмотрим некоторые аспекты практической реализации данного метода с помощью OpenGL.

Предположим, что матрица floorShadow была ранее получена нами из формулы (**LG**)**I** - **GL**. Следующий код с ее помощью строит тени для заданной плоскости:

```
/* Делаем тени полупрозрачными с использованием смешения
цветов(blending) */
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_LIGHTING);
glColor4f(0.0, 0.0, 0.0, 0.5);
glPushMatrix();
   /* Проецируем тень */
   glMultMatrixf((GLfloat *) floorShadow);
   /* Визуализируем сцену в проекции */
   RenderGeometry();
glPopMatrix();
glEnable(GL_LIGHTING);
glDisable(GL_BLEND);
/* Визуализируем сцену в обычном режиме */
RenderGeometry();
```

Матрица floorShadow может быть получена из уравнения (*) с помощью следующей функции:

```
/* параметры: plane – коэффициенты уравнения плоскости
              lightpos – координаты источника света
```

```
   возвращает: matrix - результирующая матрица
*/
void shadowmatrix(GLfloat matrix[4][4], GLfloat plane[4],
                  GLfloat lightpos[4])
{
  GLfloat dot;

  dot = plane[0] * lightpos[0] +
        plane[1] * lightpos[1] +
        plane[2] * lightpos[2] +
        plane[3] * lightpos[3];

  matrix[0][0] = dot - lightpos[0] * plane[0];
  matrix[1][0] = 0.f - lightpos[0] * plane[1];
  matrix[2][0] = 0.f - lightpos[0] * plane[2];
  matrix[3][0] = 0.f - lightpos[0] * plane[3];

  matrix[0][1] = 0.f - lightpos[1] * plane[0];
  matrix[1][1] = dot - lightpos[1] * plane[1];
  matrix[2][1] = 0.f - lightpos[1] * plane[2];
  matrix[3][1] = 0.f - lightpos[1] * plane[3];

  matrix[0][2] = 0.f - lightpos[2] * plane[0];
  matrix[1][2] = 0.f - lightpos[2] * plane[1];
  matrix[2][2] = dot - lightpos[2] * plane[2];
  matrix[3][2] = 0.f - lightpos[2] * plane[3];

  matrix[0][3] = 0.f - lightpos[3] * plane[0];
  matrix[1][3] = 0.f - lightpos[3] * plane[1];
  matrix[2][3] = 0.f - lightpos[3] * plane[2];
  matrix[3][3] = dot - lightpos[3] * plane[3];
}
```

Заметим, что тени, построенные таким образом, имеют ряд недостатков.

- Описанный алгоритм предполагает, что плоскости бесконечны, и не отрезает тени по границе. Например, если некоторый объект отбрасывает тень на стол, она не будет отсекаться по границе, и, тем более, "заворачиваться" на боковую поверхность стола.
- В некоторых местах тени может наблюдаться эффект "двойного смешения" (reblending), т.е. темные пятна в тех участках, где спроецированные треугольники перекрывают друг друга.
- С увеличением числа поверхностей сложность алгоритма резко увеличивается, т.к. для каждой поверхности нужно заново

строить всю сцену, даже если проблема отсечения теней по границе будет решена.

- Тени обычно имеют размытые границы, а в приведенном алгоритме они всегда имеют резкие края. Частично избежать этого позволяет расчет теней из нескольких источников света, расположенных рядом и последующее смешение результатов.

Имеется решение первой и второй проблемы. Для этого используется буфер маски (см. п. 6.3)

Итак, задача – отсечь вывод геометрии (тени, в данном случае) по границе некоторой произвольной области и избежать "двойного смешения". Общий алгоритм решения с использованием буфера маски таков:

1. Очищаем буфер маски значением 0
2. Отображаем заданную область отсечения, устанавливая значения в буфере маски в 1
3. Рисуем тени в тех областях, где в буфере маски установлены значения 1. Если тест проходит, устанавливаем в эти области значение 2.

Теперь рассмотрим эти этапы более подробно.

1.

```
/* очищаем буфер маски*/
glClearStencil(0x0);
/* включаем тест */
glEnable(GL_STENCIL_TEST);
```

2.

```
/* условие всегда выполнено и значение в буфере будет
равно 1*/
glStencilFunc (GL_ALWAYS, 0x1, 0xffffffff);
/* в любом случае заменяем значение в буфере маски*/
glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
/* выводим геометрию, по которой затем будет отсечена
тень*/
RenderPlane();
```

3.

```
/* условие выполнено и тест дает истину только если
значение в буфере маски равно 1 */
glStencilFunc (GL_EQUAL, 0x1, 0xffffffff);
/* значение в буфере равно 2,если тень уже выведена */
glStencilOp (GL_KEEP, GL_KEEP, GL_INCR);
/* выводим тени */
RenderShadow();
```

Строго говоря, даже при применении маскирования остаются некоторые проблемы, связанные с работой z-буфера. В частности, некоторые участки теней могут стать невидимыми. Для решения этой проблемы можно немного приподнять тени над плоскостью с помощью модификации уравнения, описывающего плоскость. Описание других методов выходит за рамки данного пособия.

7.3. Зеркальные отражения

В этом разделе мы рассмотрим алгоритм построения отражений от плоских объектов. Такие отражения придают бóльшую достоверность построенному изображению и их относительно легко реализовать.

Алгоритм использует интуитивное представление полной сцены с зеркалом как составленной из двух: «настоящей» и «виртуальной» – находящейся за зеркалом. Следовательно, процесс рисования отражений состоит из двух частей: 1) визуализации обычной сцены и 2) построения и визуализации виртуальной. Для каждого объекта «настоящей» сцены строится его отраженный двойник, который наблюдатель и увидит в зеркале.

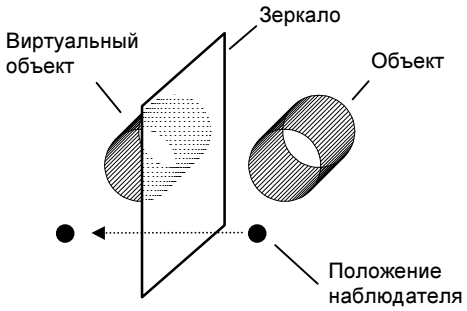


Рис. 9 Зеркальное отражение

Для иллюстрации рассмотрим комнату с зеркалом на стене. Комната и объекты, находящиеся в ней, выглядят в зеркале так, как если бы зеркало было окном, а за ним была бы еще одна такая же комната с тем же объектами, но симметрично отраженными относительно плоскости, проведенной через поверхность зеркала.

Упрощенный вариант алгоритма создания плоского отражения состоит из следующих шагов:

1. Рисуем сцену как обычно, но без объектов-зеркал.
2. Используя буфер маски, ограничиваем дальнейший вывод проекцией зеркала на экран.
3. Визуализируем сцену, отраженную относительно плоскости зеркала. При этом буфер маски позволит ограничить вывод формой проекции объекта-зеркала.

Эта последовательность действий позволит получить убедительный эффект отражения.

Рассмотрим этапы более подробно:

Сначала необходимо нарисовать сцену как обычно. Не будем останавливаться на этом этапе подробно. Заметим только, что, очищая буферы OpenGL непосредственно перед рисованием, нужно не забыть очистить буфер маски:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT|
        GL_STENCIL_BUFFER_BIT);
```

Во время визуализации сцены лучше не рисовать объекты, которые затем станут зеркальными.

На втором этапе необходимо ограничить дальнейший вывод проекцией зеркального объекта на экран.

Для этого настраиваем буфер маски и рисуем зеркало

```
glEnable(GL_STENCIL_TEST);
/* условие всегда выполнено и значение в буфере будет
равно 1*/
glStencilFunc(GL_ALWAYS, 1, 0);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

RenderMirrorObject();
```

В результате мы получили:

- в буфере кадра – корректно нарисованная сцена, за исключением области зеркала;
- в области зеркала (там, где мы хотим видеть отражение) значение буфера маски равно 1.

На третьем этапе нужно нарисовать сцену, отраженную относительно плоскости зеркального объекта.

Сначала настраиваем матрицу отражения. Матрица отражения должна зеркально отражать всю геометрию относительно плоскости, в которой лежит объект-зеркало. Ее можно получить, например, с помощью такой функции (попробуйте получить эту матрицу самостоятельно в качестве упражнения):

```
void
reflectionmatrix(GLfloat reflection_matrix[4][4],
                 GLfloat plane_point[3],
                 Glfloat plane_normal[3])
{
  GLfloat* p;
  GLfloat* v;
  float pv;

  GLfloat* p = (Glfloat*)plane_point;
  Glfloat* v = (Glfloat*)plane_normal;
  float pv = p[0]*v[0]+p[1]*v[1]+p[2]*v[2];

  reflection_matrix[0][0] = 1 – 2 * v[0] * v[0];
  reflection_matrix[1][0] = – 2 * v[0] * v[1];
  reflection_matrix[2][0] = – 2 * v[0] * v[2];
  reflection_matrix[3][0] = 2 * pv * v[0];

  reflection_matrix[0][1] = – 2 * v[0] * v[1];
  reflection_matrix[1][1] = 1– 2 * v[1] * v[1];
  reflection_matrix[2][1] = – 2 * v[1] * v[2];
  reflection_matrix[3][1] = 2 * pv * v[1];

  reflection_matrix[0][2] = – 2 * v[0] * v[2];
  reflection_matrix[1][2] = – 2 * v[1] * v[2];
  reflection_matrix[2][2] = 1 – 2 * v[2] * v[2];
  reflection_matrix[3][2] = 2 * pv * v[2];

  reflection_matrix[0][3] = 0;
  reflection_matrix[1][3] = 0;
```

```
  reflection_matrix[2][3] = 0;
  reflection_matrix[3][3] = 1;
}
```

Настраиваем буфер маски на рисование только в областях, где значения буфера равно 1:

```
/* условие выполнено и тест дает истину только если
значение в буфере маски равно 1 */
glStencilFunc (GL_EQUAL, 0x1, 0xffffffff);
/* ничего не меняем в буфере */
glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
```

и рисуем сцену еще раз (без зеркальных объектов)

```
glPushMatrix();
glMultMatrixf((float *)reflection_matrix);
RenderScene();
glPopMatrix();
```

Наконец, отключаем маскирование

```
glDisable(GL_STENCIL_TEST);
```

После этого можно опционально еще раз вывести зеркальный объект, например, с альфа-смешением – для создания эффекта замутнения зеркала и т.д.

Обратите внимание, что описанный метод корректно работает, только если за зеркальным объектом нет других объектов сцены. Поэтому существует несколько модификаций этого алгоритма, отличающихся последовательностью действий и имеющих разные ограничения на геометрию.

Контрольные вопросы

1. В результате чего возникает эффект ступенчатости изображения? Опишите алгоритм устранения ступенчатости.
2. Почему в OpenGL нет встроенной поддержки построения теней?
3. Кратко опишите предложенный метод визуализации зеркальных объектов. Почему он не работает, если за зеркалом находятся другие объекты сцены? Что будет отражаться в этом случае? Подумайте, как обойти это ограничение?

Глава 8

Оптимизация программ

8.1. Организация приложения

На первый взгляд может показаться, что производительность приложений, основанных на OpenGL, определяется в первую очередь производительностью реализации самой библиотеки OpenGL. Это верно, однако организация всего приложения также очень важна.

8.1.1. Высокоуровневая оптимизация

Обычно от программы под OpenGL требуется визуализация высокого качества на интерактивных скоростях. Но, как правило, и то и другое сразу получить не удается. Следовательно, необходим поиск компромисса между качеством и производительностью. Существует множество различных подходов, но их подробное описание выходит за пределы этого пособия. Приведем лишь несколько примеров.

- Можно отображать геометрию сцены с низким качеством во время анимации, а в моменты остановок показывать ее с наилучшим качеством. Во время интерактивного вращения (например, при нажатой клавише мыши) визуализировать модель с уменьшенным количеством примитивов. При рисовании статичного изображения отображать модель полностью.
- Аналогично, объекты, которые располагаются далеко от наблюдателя, могут быть представлены моделями пониженной сложности. Это значительно снизит нагрузку на все ступени конвейера OpenGL. Объекты, которые находятся полностью вне поля видимости, могут быть эффективно отсечены без передачи на конвейер OpenGL с помощью проверки попадания ограничивающих их простых объемов (сфер или кубов) в пирамиду зрения.
- Во время анимации можно отключить псевдотонирование (dithering), плавную заливку, текстуры. Опять-таки включать все это во время демонстрации статичных изображений. Этот

подход особенно эффективен для систем без аппаратной поддержки OpenGL.

8.1.2. Низкоуровневая оптимизация

Объекты, отображаемые с помощью OpenGL, хранятся в некоторых структурах данных. Одни типы таких структур более эффективны в использовании, чем другие, что определяет скорость визуализации.

Желательно, чтобы использовались структуры данных, которые могут быть быстро и эффективно переданы на конвейер OpenGL. Например, если мы хотим отобразить массив треугольников, то использование указателя на этот массив значительно более эффективно, чем передача его OpenGL поэлементно.

Пример:

Предположим, что мы пишем приложение, которое реализует рисование карты местности. Один из компонентов базы данных – список городов с их шириной, долготой и названием. Соответствующая структура данных может быть такой:

```
struct city
{
   float latitute, longitude; /* положение города */
   char *name;                /* название  */
   int large_flag;      /* 0 = маленький, 1 = большой */
};
```

Список городов может храниться как массив таких структур. Допустим, мы пишем функцию, которая рисует города на карте в виде точек разного размера с подписями:

```
void draw_cities( int n, struct city citylist[] )
{
   int i;
   for (i=0; i < n; i++)
     {
     if (citylist[i].large_flag)
         glPointSize( 4.0 );
     else
         glPointSize( 2.0 );

     glBegin( GL_POINTS );
     glVertex2f( citylist[i].longitude,
```

```
                citylist[i].latitude );
    glEnd();
    /* рисуем название города */
    DrawText(citylist[i].longitude,
             citylist[i].latitude,
             citylist[i].name);
   }
}
```

Это реализация неудачна по следующим причинам:

- `glPointSize` вызывается для каждой итерации цикла.
- между `glBegin` и `glEnd` рисуется только одна точка.
- вершины определяются в неоптимальном формате.

Ниже приведено более рациональное решение:

```
void draw_cities( int n, struct city citylist[] )
{
    int i;
    /* сначала рисуем маленькие точки */
    glPointSize( 2.0 );
    glBegin( GL_POINTS );
    for (i=0; i < n ;i++)
    {
       if (citylist[i].large_flag==0) {
          glVertex2f( citylist[i].longitude,
                      citylist[i].latitude );
      }
    }
    glEnd();
    /* большие точки рисуем во вторую очередь */
    glPointSize( 4.0 );
    glBegin( GL_POINTS );
    for (i=0; i < n ;i++)
      {
      if (citylist[i].large_flag==1)
      {
        glVertex2f( citylist[i].longitude,
                    citylist[i].latitude );
      }
    }
    glEnd();
   /* затем рисуем названия городов */
 for (i=0; i < n ;i++)
 {
```

```
      DrawText(citylist[i].longitude,
               citylist[i].latitude,
               citylist[i].name);
    }
}
```

В такой реализации мы вызываем `glPointSize` дважды и увеличиваем число вершин между `glBegin` и `glEnd`.

Однако остаются еще пути для оптимизации. Если мы поменяем наши структуры данных, то можем еще повысить эффективность рисования точек. Например:

```
struct city_list
{
  int num_cities; /* число городов в списке */
  float *position;/* координаты города */
  char **name;    /* указатель на названия городов */
  float size;     /* размер точки, обозначающей город */
};
```

Теперь города разных размеров хранятся в разных списках. Положения точек хранятся отдельно в динамическом массиве. После реорганизации мы исключаем необходимость в условном операторе внутри `glBegin/glEnd` и получаем возможность использовать массивы вершин для оптимизации. В результате наша функция выглядит следующим образом:

```
void draw_cities( struct city_list *list )
  {
  int i;

  /* рисуем точки */
  glPointSize( list->size );

  glVertexPointer( 2, GL_FLOAT, 0,
                   list->num_cities,
                   list->position );
  glDrawArrays( GL_POINTS, 0, list->num_cities );
  /* рисуем название города */
  for (i=0; i < list->num_cities ;i++)
  {
     DrawText(citylist[i].longitude,
              citylist[i].latitude
              citylist[i].name);
  }
}
```

8.2. Оптимизация вызовов OpenGL

Существует много возможностей улучшения производительности OpenGL. К тому же разные подходы к оптимизации приводят к разным эффектам при аппаратной и программной визуализации. Например, интерполяция цветов может быть очень дорогой операцией без аппаратной поддержки, а при аппаратной визуализации почти не дает задержек.

После каждой из следующих методик следуют квадратные скобки, в которых указан один из символов, обозначающих значимость данной методики для конкретной системы

- [А] – предпочтительно для систем с аппаратной поддержкой OpenGL
- [П] – предпочтительно для программных реализаций
- [все] – вероятно предпочтительно для всех реализаций

8.2.1. Передача данных в OpenGL

В данном разделе рассмотрим способы минимизации времени на передачу данных о примитивах в OpenGL

Используйте связанные примитивы.

Связанные примитивы, такие как **GL_LINES**, **GL_LINE_LOOP**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLE_FAN**, и **GL_QUAD_STRIP** требуют для определения меньше вершин, чем отдельные линия или многоугольник. Это уменьшает количество данных, передаваемых OpenGL [все]

Используйте массивы вершин.

На большинстве архитектур замена множественных вызовов `glVertex/glColor/glNormal` на механизм массивов вершин может быть очень выигрышной. [все]

Используйте индексированные примитивы.

В некоторых случаях даже при использовании связанных примитивов **GL_TRIANGLE_STRIP** (**GL_QUAD_STRIP**) вершины дублируются.

Чтобы не передавать в OpenGL дубли, увеличивая нагрузку на шину, используйте команду `glDrawElements()` (см. раздел 2.5.) [все]

Задавайте необходимые массивы одной командой.

Вместо использования команд

`glVertexPointer/glColorPointer/glNormalPointer`
лучше пользоваться одной командой

```
void glInterleavedArrays ( Glint format,
                           Glsizei stride,
                           void * ptr);
```

так, если имеется структура

```
typedef struct tag_VERTEX_DATA
       {
       float color[4];
       float normal[3];
       float vertex[3];
       }VERTEX_DATA;
VERTEX_DATA * pData;
```

то параметры можно передать с помощью следующей команды

```
glInterleavedArrays (GL_C4F_N3F_V3F, 0, pData);
```

что означает, что первые четыре float относятся к цвету, затем три float к нормали, и последние три float задают координаты вершины. Более подробное описание команды смотрите в спецификации OpenGL. [все]

Храните данные о вершинах в памяти последовательно.

Последовательное расположение данных в памяти улучшает скорость обмена между основной памятью и графической подсистемой. [A]

Используйте векторные версии `glVertex`, `glColor`, `glNormal` и `glTexCoord`.

Функции `glVertex*()`, `glColor*()` и т.д., которые в качестве аргументов принимают указатели (например, `glVertex3fv(v)`) могут работать значительно быстрее, чем их соответствующие версии `glVertex3f(x,y,z)` [все]

Уменьшайте сложность примитивов.

Во многих случаях будьте внимательны, чтобы не разбивать большие плоскости на части сильнее, чем необходимо. Поэкспериментируйте, например, с примитивами GLU для определения наилучшего соотношения качества и производительности. Текстурированные объекты, например, могут быть качественно отображены с небольшой сложностью геометрии. [все]

Используйте дисплейные списки.

Используйте дисплейные списки для наиболее часто выводимых объектов. Дисплейные списки могут храниться в памяти графической подсистемы и, следовательно, исключать частые перемещения данных из основной памяти. [A]

Не указывайте ненужные атрибуты вершин

Если освещение выключено, не вызывайте `glNormal`. Если не используются текстуры, не вызывайте `glTexCoord`, и т.д. [все]

Минимизируйте количество лишнего кода между `glBegin/glEnd`

Для максимальной производительности на high-end системах важно, чтобы информация о вершинах была передана графической подсистеме максимально быстро. Избегайте лишнего кода между `glBegin/glEnd`.

Пример неудачного решения:

```
glBegin(GL_TRIANGLE_STRIP);
for (i=0; i < n; i++)
{
   if (lighting)
   {
      glNormal3fv(norm[i]);
   }
   glVertex3fv(vert[i]);
}
glEnd();
```

Эта конструкция плоха тем, что мы проверяем переменную `lighting` перед каждой вершиной. Этого можно избежать, за счет частичного дублирования кода:

```
if (lighting)
```

```
{
   glBegin(GL_TRIANGLE_STRIP);
   for (i=0; i < n ;i++)
   {
      glNormal3fv(norm[i]);
      glVertex3fv(vert[i]);
   }
   glEnd();
 }
 else
 {
   glBegin(GL_TRIANGLE_STRIP);
   for (i=0; i < n ;i++)
   {
      glVertex3fv(vert[i]);
   }
   glEnd();
}
```

8.2.2. Преобразования

Преобразования включают в себя трансформации вершин от координат, указанных в `glVertex*()`, к оконным координатам, отсечение, освещение и т.д.

Освещение

- Избегайте использования локальных источников света, т.е. координаты источника должны быть в форме (x,y,z,0) [П]
- Избегайте использования точечных источников света. [все]
- Избегайте использования двухстороннего освещения (two-sided lighting). [все]
- Избегайте использования отрицательные коэффициентов в параметрах материала и цвета. [П]
- Избегайте использования локальной модели освещения .[все]
- Избегайте частой смены параметра материала **GL_SHININESS** [П]
- Некоторые реализации OpenGL оптимизированы для случая одного источника света [А, П]

- Рассмотрите возможность заранее просчитать освещение. Можно получить эффект освещения, задавая цвета вершин вместо нормалей. [все]

Отключайте нормализацию векторов нормалей, когда это не необходимо.

Команда `glEnable/Disable(GL_NORMALIZE)` управляет нормализацией векторов нормалей перед использованием. Если вы не используете команду `glScale`, то нормализацию можно отключить без посторонних эффектов. По умолчанию эта опция выключена. [все]

Используйте связанные примитивы.

Связанные примитивы, такие как **GL_LINES**, **GL_LINE_LOOP**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLE_FAN**, и **GL_QUAD_STRIP** уменьшают нагрузку на конвейер OpenGL, а также уменьшают количество данных, передаваемых графической подсистеме.

8.2.3. Растеризация

Растеризация часто является узким местом программных реализаций OpenGL.

Отключайте интерполяцию цветов, когда в этом нет необходимости

Интерполяция цветов включена по умолчанию. Плоское затенение не требует интерполяции четырех компонент цвета и, как правило, быстрее на программных реализациях OpenGL. Аппаратные реализации обычно выполняют оба вида затенения с одинаковой скоростью. Для отключения используйте команду `glShadeModel(GL_FLAT)` [П]

Отключайте тест на глубину, когда в этом нет необходимости.

Фоновые объекты, например, могут быть нарисованы без теста на глубину, если они визуализируется первыми [все]

Используйте отсечение обратных граней полигонов

Замкнутые объекты могут быть нарисованы с установленным режимом отсечения обратных граней `glEnable(GL_CULL_FACE)` Иногда это позволяет отбросить до половины многоугольников, не растеризуя их.[все]

Избегайте лишних операций с пикселями

Маскирование, альфа-смешивание и другие попиксельные операции могут занимать существенное время на этапе растеризации. Отключайте все операции, которые вы не используете. [все]

Уменьшайте размер окна или разрешение экрана

Простой способ уменьшить время растеризации – уменьшить число пикселей, которые будут нарисованы. Если меньшие размеры окна или меньшее разрешение экрана приемлемы, то это хороший путь для увеличения скорости растеризации. [все]

8.2.4. Текстурирование

Наложение текстур является дорогой операцией, как в программных, так и в аппаратных реализациях.

Используйте эффективные форматы хранения изображений

Формат GL_UNSIGNED_BYTE обычно наиболее всего подходит для передачи текстуры в OpenGL. [все]

Объединяйте текстуры в текстурные объекты или дисплейные списки.

Это особенно важно, если вы используете несколько текстур и позволяет графической подсистеме эффективно управлять размещением текстур в видеопамяти. [все]

Не используйте текстуры большого размера

Небольшие текстуры быстрее обрабатываются и занимают меньше памяти, что позволят хранить сразу несколько текстур в памяти графической подсистемы [all]

Комбинируйте небольшие текстуры в одну

Если вы используете несколько маленьких текстур, можно объединить их в одну большего размера и изменить текстурные координаты для работы нужной подтекстурой. Это позволяет уменьшить число переключений текстур.

Анимированные текстуры

Если вы хотите использовать анимированные текстуры, не используйте команду `glTexImage2D` чтобы обновлять образ

текстуры. Вместо этого используйте `glTexSubImage2D` или `glTexCopyTexSubImage2D`.

8.2.5. Очистка буферов

Очистка буферов цвета, глубины, маски и буфера-накопителя может требовать много времени, особенно в программных реализациях OpenGL. В этом разделе описаны некоторые приемы, которые могут помочь оптимизировать эту операцию.

Используйте команду `glClear` с осторожностью [все]

Очищайте все нужные буферы с помощью одной команды `glClear`.

Неверно:

```
glClear(GL_COLOR_BUFFER_BIT);
if (stenciling)  /* очистить буфер маски? */
{
    glClear(GL_STENCIL_BUFFER_BIT);
}
```

Верно:

```
if (stenciling) /* очистить буфер маски? */
{
   glClear(GL_COLOR_BUFFER_BIT |
           STENCIL_BUFFER_BIT);
}
else
{
  glClear(GL_COLOR_BUFFER_BIT);
}
```

Отключайте размывание (dithering)

Отключайте размывание перед очисткой буфера. Обычно различие между очистками с включенным размыванием и без него незаметно. [П]

Используйте ножницы (scissors) для очистки меньшей области

Если вы не хотите очищать весь буфер, используйте `glScissor()` для ограничения очистки по заданной области [все].

Не очищайте буфер цвета полностью

Если ваша сцена занимает только часть окна, нет необходимости очищать весь буфер цвета. [П]

Избегайте команды `glClearDepth(d)`, где d!=1.0

Некоторые программные реализации оптимизированы для очистки буфера с глубиной 1.0. [П]

8.2.6. Разное

Проверяйте ошибки GL во время написания программ. [все]

Вызывайте команду `glGetError()` для проверки, не произошла ли ошибки во время вызова одной из функций OpenGL. Как правило, ошибки возникают из-за неверных параметров команд OpenGL или неверной последовательности команд. Для финальных версий кода отключайте эти проверки, так как они могут существенно замедлить работу. Для проверки можно использовать, например, такой макрос:

```
#include <assert.h>
#define CHECK_GL \
   assert(glGetError() != GL_NO_ERROR);
```

Использовать его можно так:

```
glBegin(GL_TRIANGLES);
glVertex3f(1,1,1);
glEnd();
CHECK_GL
```

Используйте `glColorMaterial` вместо `glMaterial`

Если в сцене материалы объектов различаются лишь одним параметром, команда `glColorMaterial` может быть быстрее, чем `glMaterial` [все]

Минимизируйте число изменений состояния OpenGL

Команды, изменяющие состояние OpenGL (`glEnable/glDisable/glBind`Texture и другие), вызывают повторные внутренние проверки целостности, создание дополнительных структур данных и т.д., что может приводить к задержкам [все]

Избегайте использования команды `glPolygonMode`

Если вам необходимо рисовать много незакрашенных многоугольников, используйте `glBegin` с **GL_POINTS**, **GL_LINES**, **GL_LINE_LOOP** или **GL_LINE_STRIP** вместо изменения режима рисования примитивов, так как это может быть намного быстрее [все]

Конечно, эти рекомендации охватывают лишь малую часть возможностей по оптимизации OpenGL-приложений. Тем не менее, при их правильном использовании можно достичь существенного ускорения работы ваших программ.

Контрольные вопросы

1. Перечислите известные вам методы высокоуровневой оптимизации OpenGL-приложений
2. Почему предпочтительнее использование связанных примитивов?
3. Какая из двух команд выполняется OpenGL быстрее?

```
glVertex3f(1,1,1)
```

или

```
float vct[3] = {1,1,1};
glVertex3fv(vct)
```

Приложение А.

Структура GLUT-приложения

Далее будем рассматривать построение консольного приложения при помощи библиотеки GLUT. Эта библиотека обеспечивает единый интерфейс для работы с окнами вне зависимости от платформы, поэтому описываемая ниже структура приложения остается неизменной для операционных систем Windows, Linux и других.

Функции GLUT могут быть классифицированы на несколько групп по своему назначению:

- Инициализация
- Начало обработки событий
- Управление окнами
- Управление меню
- Регистрация функций с обратным вызовом
- Управление индексированной палитрой цветов
- Отображение шрифтов
- Отображение дополнительных геометрических фигур (тор, конус и др.)

Инициализация проводится с помощью функции:

```
glutInit (int *argcp, char **argv)
```

Переменная *argcp* есть указатель на стандартную переменную argc описываемую в функции main(), а *argv* – указатель на параметры, передаваемые программе при запуске, который описывается там же. Эта функция проводит необходимые начальные действия для построения

окна приложения, и только несколько функций GLUT могут быть вызваны до нее. К ним относятся:

```
glutInitWindowPosition  (int x, int y)
glutInitWindowSize  (int width, int height)
glutInitDisplayMode  (unsigned int mode)
```

Первые две функции задают соответственно положение и размер окна, а последняя функция определяет различные режимы отображения информации, которые могут совместно задаваться с использованием операции побитового “или” (“ | “) :

GLUT_RGBA	Режим RGBA. Используется по умолчанию, если не указаны явно режимы **GLUT_RGBA** или **GLUT_INDEX**.
GLUT_RGB	То же, что и **GLUT_RGBA**.
GLUT_INDEX	Режим индексированных цветов (использование палитры). Отменяет **GLUT_RGBA**.
GLUT_SINGLE	Окно с одиночным буфером. Используется по умолчанию.
GLUT_DOUBLE	Окно с двойным буфером. Отменяет **GLUT_SINGLE**.
GLUT_STENCIL	Окно с буфером маски.
GLUT_ACCUM	Окно с буфером-накопителем.
GLUT_DEPTH	Окно с буфером глубины.

Это неполный список параметров для данной функции, однако для большинства случаев этого бывает достаточно.

Работа с буфером маски и буфером накопления описана в главе 6.

Функции библиотеки GLUT реализуют так называемый событийно-управляемый механизм. Это означает, что есть некоторый внутренний цикл, который запускается после соответствующей инициализации и обрабатывает одно за другим все события, объявленные во время инициализации. К событиям относятся: щелчок мыши, закрытие окна, изменение свойств окна, передвижение курсора, нажатие клавиши, и "пустое" (idle) событие, когда ничего не происходит. Для проведения

периодической проверки совершения того или иного события надо зарегистрировать функцию, которая будет его обрабатывать. Для этого используются функции вида:

```
void glutDisplayFunc  (void (*func) (void))
void glutReshapeFunc (void (*func) (int width,
                                    int height))
void glutMouseFunc  (void (*func) (int button,
                     int state, int x, int y))
void glutIdleFunc (void (*func) (void))
void glutMotionFunc (void (*func)(int x, int y));
void glutPassiveMotionFunc (void (*func)(int x, int y));
```

Параметром для них является имя соответствующей функции заданного типа. С помощью **glutDisplayFunc()** задается функция рисования для окна приложения, которая вызывается при необходимости создания или восстановления изображения. Для явного указания, что окно надо обновить, иногда удобно использовать функцию

```
void glutPostRedisplay (void)
```

Через **glutReshapeFunc()** устанавливается функция обработки изменения размеров окна пользователем, которой передаются новые размеры.

glutMouseFunc() определяет функцию – обработчик команд от мыши, а **glutIdleFunc()** задает функцию, которая будет вызываться каждый раз, когда нет событий от пользователя.

Функция, определяемая **glutMotionFunc** вызывается, когда пользователь двигает указатель мыши, удерживая нажатой кнопку мыши. **glutPassiveMotionFunc** регистрирует функцию, вызываемую, если пользователь двигает указатель мыши и *не* нажато ни одной кпопки мыши.

Контроль всех событий происходит внутри бесконечного цикла в функции

```
void  glutMainLoop  (void)
```

которая обычно вызывается в конце любой программы, использующей GLUT. Структура приложения, использующего анимацию, будет следующей:

```
 #include <GL/glut.h>

 void MyIdle(void)
 {
 /*Код, который меняет переменные, определяющие следующий
кадр */
…
};

void MyDisplay(void)
{
 /* Код OpenGL, который отображает кадр */
  …
 /* После рисования переставляем  буферы */
  glutSwapBuffers();
 };


void main(int argcp, char **argv)
{
 /* Инициализация GLUT */
 glutInit(&argcp, argv);
 glutInitWindowSize(640, 480);
 glutInitWindowPosition(0, 0);
 /* Открытие окна */
 glutCreateWindow("My OpenGL Application");
 /* Выбор режима:  двойной буфер и RGBA цвета */
 glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
GLUT_DEPTH);
 /* Регистрация вызываемых функций */
 glutDisplayFunc(MyDisplay);
 glutIdleFunc(MyIdle);
 /* Запуск механизма обработки событий */
 glutMainLoop();
};
```

В случае , если приложение должно строить статичное изображение, можно заменить **GLUT_DOUBLE** на **GLUT_SINGLE**, так как одного буфера в этом случае будет достаточно, и убрать вызов функции *glutIdleFunc().*

Приложение В.

Примитивы библиотек GLU и GLUT

Рассмотрим стандартные команды построения примитивов, которые реализованы в библиотеках GLU и GLUT.

Чтобы построить примитив из библиотеки GLU, надо сначала создать указатель на quadric-объект с помощью команды `gluNewQuadric()`, а затем вызвать одну из команд `gluSphere()`, `gluCylinder()`, `gluDisk()`, `gluPartialDisk()`. Рассмотрим эти команды отдельно:

```
void gluSphere (GLUquadricObj  *qobj, GLdouble radius,
                GLint slices, GLint stacks)
```

Эта функция строит сферу с центром в начале координат и радиусом *radius*. При этом число разбиений сферы вокруг оси z задается параметром *slices*, а вдоль оси z – параметром *stacks*.

```
void gluCylinder (GLUquadricObj  *qobj,
                  GLdouble baseRadius,
                  GLdouble topRadius,
                  GLdouble height, GLint slices,
                  GLint stacks)
```

Данная функция строит цилиндр без оснований (кольцо), продольная ось параллельна оси z, заднее основание имеет радиус *baseRadius*, и расположено в плоскости z=0, переднее основание имеет радиус *topRadius* и расположено в плоскости z= *height*. Если задать один из радиусов равным нулю, то будет построен конус. Параметры *slices* и *stacks* имеют аналогичный смысл, что и в предыдущей команде.

```
void gluDisk (GLUquadricObj  *qobj,
              GLdouble innerRadius,
              GLdouble outerRadius,
              GLint slices,
              GLint loops)
```

Функция строит плоский диск (круг) с центром в начале координат и радиусом *outerRadius*. При этом если значение *innerRadius* отлично от нуля, то в центре диска будет находиться отверстие радиусом *innerRadius*. Параметр *slices* задает число разбиений диска вокруг оси z, а параметр *loops* – число концентрических колец, перпендикулярных оси z.

```
void gluPartialDisk (GLUquadricObj *qobj,
                     GLdouble innerRadius,
                     GLdouble outerRadius,
                     GLint slices,
                     GLint loops,
                     GLdouble startAngle,
                     GLdouble sweepAngle);
```

Отличие этой команды от предыдущей заключается в том, что она строит сектор круга, начальный и конечный углы которого отсчитываются против часовой стрелки от положительного направления оси y и задаются параметрами *startAngle* и *sweepAngle*. Углы измеряются в градусах.

Команды, проводящие построение примитивов из библиотеки GLUT, реализованы через стандартные примитивы OpenGL и GLU. Для построения нужного примитива достаточно произвести вызов соответствующей команды.

```
void glutSolidSphere  (GLdouble radius,
                       GLint slices,
                       GLint stacks)
void glutWireSphere   (GLdouble radius,
                       GLint slices,
                       GLint stacks)
```

Команда **glutSolidSphere()** строит сферу, а **glutWireSphere()** – каркас сферы радиусом *radius*. Остальные параметры те же, что и в предыдущих командах.

```
void glutSolidCube (GLdouble size)
void glutWireCube  (GLdouble size)
```

Команды строят куб или каркас куба с центром в начале координат и длиной ребра *size*.

```
void glutSolidCone  (GLdouble base, GLdouble height,
                     GLint slices, GLint stacks)
void glutWireCone  (GLdouble base, GLdouble height,
                    GLint slices, GLint stacks)
```

Эти команды строят конус или его каркас высотой *height* и радиусом основания *base*, расположенный вдоль оси z. Основание находится в плоскости z=0.

```
void glutSolidTorus (GLdouble innerRadius,
                     GLdouble outerRadius,
                     GLint nsides,
                     GLint rings)
void glutWireTorus  (GLdouble innerRadius,
                     GLdouble outerRadius,
                     GLint nsides,
                     GLint rings)
```

Эти команды строят тор или его каркас в плоскости z=0. Внутренний и внешний радиусы задаются параметрами *innerRadius*, *outerRadius*. Параметр *nsides* задает число сторон в кольцах, составляющих ортогональное сечение тора, а *rings* – число радиальных разбиений тора.

```
void glutSolidTetrahedron  (void)
void glutWireTetrahedron  (void)
```

Эти команды строят тетраэдр (правильную треугольную пирамиду) или его каркас, при этом радиус описанной сферы вокруг него равен 1.

```
void glutSolidOctahedron  (void)
void glutWireOctahedron  (void)
```

Эти команды строят октаэдр или его каркас, радиус описанной вокруг него сферы равен 1.

```
void glutSolidDodecahedron  (void)
void glutWireDodecahedron  (void)
```

Эти команды строят додекаэдр или его каркас, радиус описанной вокруг него сферы равен квадратному корню из трех.

```
void glutSolidIcosahedron  (void)
void glutWireIcosahedron  (void)
```

Эти команды строят икосаэдр или его каркас, радиус описанной вокруг него сферы равен 1.

Для корректного построения перечисленных примитивов необходимо удалять невидимые линии и поверхности, для чего надо включить соответствующий режим вызовом команды **glEnable**(GL_DEPTH_TEST).

Приложение С.

Настройка приложений OpenGL

С.1. Создание приложения в среде Borland C++ 5.02

Вначале необходимо обеспечить наличие файлов glut.h, glut32.lib, glut32.dll в каталогах BorlandC\Include\Gl, BorlandC\Lib, Windows\System соответственно. Также в этих каталогах надо проверить наличие файлов gl.h, glu.h, opengl32.lib, glu32.lib, opengl32.dll, glu32.dll, которые обычно входят в состав BorlandC++ и Windows. При этом надо учитывать, что версии Microsoft файлов opengl32.lib, glu32.lib, glut32.lib для Borland C++ не подходят, и следует использовать только совместимые версии. Чтобы создать такие версии, надо использовать стандартную программу ‘implib’, которая находится в каталоге BorlandC\Bin. Для этого надо выполнить команды вида

```
implib  BorlandC\Lib\filename.lib  filename.dll
```

для перечисленных файлов, которые создают нужный *.lib файл из соответствующего *.dll файла. Кроме того, надо отметить, что компилятор BorlandC не может по неизвестным причинам использовать файл glaux.lib, входящий в состав BorlandC++5.02, при компиляции приложения, использующего библиотеку GLAUX, поэтому возможно от этой библиотеки придется отказаться. Для создания приложения надо выполнить следующие действия:

- Создание проекта: для этого надо выбрать *Project→New Project* и заполнить поля в окне *Target Expert* следующим образом: в поле *Platform* выбрать Win32, в поле *Taget Model* выбрать *Console*, нажать *Advanced* и отменить выбор пунктов ‘ *.rc ‘ и ‘ *.def ‘.
- Подключить к проекту библиотеки OpenGL. Для этого надо выбрать в окне проекта название исполняемого файла проекта (*.exe) и, нажав правую кнопку мыши, выбрать в контекстном меню пункт *Add node*. Затем надо определить положение файлов opengl32.lib, glu32.lib, glut32.lib.

- Для компиляции выбрать *Project→Build All*, для выполнения – *Debug→Run*.

C.2. Создание приложения в среде MS Visual C++ 6.0

Перед началом работы необходимо скопировать файлы glut.h, glut32.lib glut32.dll в каталоги MSVC\Include\Gl, MSVC\Lib, Windows\System соответственно. Также в этих каталогах надо проверить наличие файлов gl.h, glu.h, opengl32.lib, glu32.lib, opengl32.dll, glu32.dll, которые обычно входят в состав Visual C++ и Windows. При использовании команд из библиотеки GLAUX к перечисленным файлам надо добавить glaux.h, glaux.lib.

Для создания приложения надо выполнить следующие действия:

- Создание проекта: для этого надо выбрать *File→New→Projects→Win32 Console Application*, набрать имя проекта, OK.
- В появившемся окне выбрать '*An empty project*', Finish,OK.
- Текст программы можно либо разместить в созданном текстовом файле (выбрав *File→New→Files→Text File*), либо добавиьб файл с расширением *.c или *.cpp в проект (выбрав *Project→Add To Project→Files*).
- Подключить к проекту библиотеки OpenGL. Для этого надо выбрать *Project→Settings→Link* и в поле *Object/library modules* набрать названия нужных библиотек: opengl32.lib, glu32.lib, glut32.lib и, если надо, glaux.lib.
- Для компиляции выбрать *Build→Build program.exe*, для выполнения – *Build→Execute program.exe*.
- Чтобы при запуске не появлялось текстовое окно, надо выбрать *Project→Settings→Link* и в поле *Project Options* вместо 'subsystem:console' набрать 'subsystem:windows', и набрать там же строку '/entry:mainCRTStartup'
- Когда программа готова, рекомендуется перекомпилировать ее в режиме 'Release' для оптимизации по быстродействию и объему. Для этого сначала надо выбрать *Build→Set Active Configuration…* и отметить '…-Win32 Release', а затем заново подключить необходимые библиотеки.

C.3. Создание приложения в среде Borland C++ Builder 6.

Перед началом работы необходимо скопировать файлы glut.h, glut32.lib glut32.dll в каталоги CBuilder6\Include\Gl, CBuilder6\Lib, Windows\System соответственно. Также в этих каталогах надо проверить наличие файлов gl.h, glu.h, opengl32.lib, glu32.lib, opengl32.dll, glu32.dll, которые обычно входят в состав Borland C++ и Windows.

При этом надо учитывать, что версии Microsoft файла glut32.lib для Borland C++ Builder 6 не подходят, и следует использовать только совместимую версию. Чтобы создать такую версию, надо использовать стандартную программу 'implib', которая находится в каталоге CBuilder6\Bin. Для этого надо выполнить команду

```
implib  glut32.lib  glut32.dll
```

, которая создает нужный lib-файл из соответствующего dll-файла.

Для создания приложения надо выполнить следующие действия:

- Создание проекта: для этого надо выбрать *File→New→Other→Console Wizard*, OK.
- В появившемся окне выбрать Source Type – C++, Console Application, сбросить опции 'Use VCL', 'Use CLX', 'Multi Threaded'. Нажать OK.
- Текст программы можно либо разместить в созданном текстовом файле, либо удалить его из проекта (*Project→Remove From Project*) и добавить файл с расширением *.c или *.cpp в проект (выбрав *Project→Add To Project*).
- Сохраните созданный проект в желаемом каталоге (выбрав *File→Save All*).
- Подключить к проекту библиотеку GLUT. Для этого надо выбрать *Project→Add To Project* и добавить файл glut32.lib
- Для компиляции выбрать *Project→Build …*, для выполнения – *Run→Run*.
- Когда программа готова, рекомендуется перекомпилировать ее в режиме 'Release' для оптимизации по быстродействию и объему. Для этого сначала надо выбрать *Project→Options→Compiler* и нажать кнопку 'Release'.

Приложение D.

Демонстрационные программы

D.1. Пример 1: Простое GLUT-приложение

Этот простой пример предназначен для демонстрации структуры GLUT-приложения и простейших основ OpenGL. Результатом работы программы является случайный набор цветных прямоугольников, который меняется при нажатии левой кнопки мыши. С помощью правой кнопки мыши можно менять режим заливки прямоугольников.

```
#include <stdlib.h>
#include <gl/glut.h>

#ifdef random
#undef random
#endif

#define random(m) (float)rand()*m/RAND_MAX

/* ширина и высота окна */
GLint Width = 512, Height = 512;
/* число прямоугольников в окне */
int   Times = 100;
/* с заполнением ?*/
int   FillFlag  = 1;

long  Seed = 0;

/* функция отображает прямоугольник */
void
DrawRect( float x1, float y1, float x2, float y2,
          int FillFlag )
{
  glBegin(FillFlag ? GL_QUADS : GL_LINE_LOOP);
  glVertex2f(x1, y1);
  glVertex2f(x2, y1);
```

```
  glVertex2f(x2, y2);
  glVertex2f(x1, y2);
  glEnd();
}

/* управляет всем выводом на экран */
void
Display(void)
{
  int i;
  float x1, y1, x2, y2;
  float r, g, b;

  srand(Seed);

  glClearColor(0, 0, 0, 1);
  glClear(GL_COLOR_BUFFER_BIT);

  for( i = 0; i < Times; i++ ) {
    r = random(1);
    g = random(1);
    b = random(1);
    glColor3f( r, g, b );

    x1 = random(1) * Width;
    y1 = random(1) * Height;
    x2 = random(1) * Width;
    y2 = random(1) * Height;
    DrawRect(x1, y1, x2, y2, FillFlag);
  }

  glFinish();
}

/* Вызывается при изменении размеров окна */
void
Reshape(GLint w, GLint h)
{
  Width = w;
  Height = h;

  glViewport(0, 0, w, h);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0, w, 0, h, -1.0, 1.0);
```

```
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

/* Обрабатывает сообщения от мыши */
void
Mouse(int button, int state, int x, int y)
{
  if( state == GLUT_DOWN ) {
    switch( button ) {
      case GLUT_LEFT_BUTTON:
        Seed = random(RAND_MAX);
        break;
      case GLUT_RIGHT_BUTTON:
        FillFlag = !FillFlag;
        break;
    }
    glutPostRedisplay();
  }
}

/* Обрабатывает сообщения от клавиатуры */
void
Keyboard( unsigned char key, int x, int y )
{
#define ESCAPE '\033'

  if( key == ESCAPE )
      exit(0);
}

main(int argc, char *argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB);
  glutInitWindowSize(Width, Height);
  glutCreateWindow("Rect draw example (RGB)");

  glutDisplayFunc(Display);
  glutReshapeFunc(Reshape);
  glutKeyboardFunc(Keyboard);
  glutMouseFunc(Mouse);

  glutMainLoop();
}
```

D.2. Пример 2: Модель освещения OpenGL

Программа предназначена для демонстрации модели освещения OpenGL на примере простой сцена, состоящей из тора, конуса и шара. Объектам назначаются разные материалы. В сцене присутствует точечный источник света.

```
#include <GL/glut.h>
#include <stdlib.h>

/* параметры материала тора */
float mat1_dif[]={0.8f,0.8f,0.0f};
float mat1_amb[]= {0.2f,0.2f,0.2f};
float mat1_spec[]={0.6f,0.6f,0.6f};
float mat1_shininess=0.5f*128;

/* параметры материала конуса */
float mat2_dif[]={0.0f,0.0f,0.8f};
float mat2_amb[]= {0.2f,0.2f,0.2f};
float mat2_spec[]={0.6f,0.6f,0.6f};
float mat2_shininess=0.7f*128;

/* параметры материала шара */
float mat3_dif[]={0.9f,0.2f,0.0f};
float mat3_amb[]= {0.2f,0.2f,0.2f};
float mat3_spec[]={0.6f,0.6f,0.6f};
float mat3_shininess=0.1f*128;

/* Инициализируем параметры материалов и
 * источника света
 */
void init (void)
{
   GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

   /* устанавливаем параметры источника света */
   glLightfv (GL_LIGHT0, GL_AMBIENT, light_ambient);
   glLightfv (GL_LIGHT0, GL_DIFFUSE, light_diffuse);
   glLightfv (GL_LIGHT0, GL_SPECULAR, light_specular);
   glLightfv (GL_LIGHT0, GL_POSITION, light_position);

   /* включаем освещение и источник света */
```

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

   /* включаем z-буфер */
   glEnable(GL_DEPTH_TEST);

}

/* Функция вызывается при необходимости
 * перерисовки изображения. В ней осуществляется
 * весь вывод геометрии.
 */
void display (void)
{
   /* очищаем буфер кадра и буфер глубины */
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix ();
   glRotatef (20.0, 1.0, 0.0, 0.0);

   /* отображаем тор */
   glMaterialfv (GL_FRONT,GL_AMBIENT,mat1_amb);
   glMaterialfv (GL_FRONT,GL_DIFFUSE,mat1_dif);
   glMaterialfv (GL_FRONT,GL_SPECULAR,mat1_spec);
   glMaterialf  (GL_FRONT,GL_SHININESS,mat1_shininess);

   glPushMatrix ();
   glTranslatef (-0.75, 0.5, 0.0);
   glRotatef (90.0, 1.0, 0.0, 0.0);
   glutSolidTorus (0.275, 0.85, 15, 15);
   glPopMatrix ();

   /* отображаем конус */
   glMaterialfv (GL_FRONT,GL_AMBIENT,mat2_amb);
   glMaterialfv (GL_FRONT,GL_DIFFUSE,mat2_dif);
   glMaterialfv (GL_FRONT,GL_SPECULAR,mat2_spec);
   glMaterialf  (GL_FRONT,GL_SHININESS,mat2_shininess);

   glPushMatrix ();
   glTranslatef (-0.75, -0.5, 0.0);
   glRotatef (270.0, 1.0, 0.0, 0.0);
   glutSolidCone (1.0, 2.0, 15, 15);
   glPopMatrix ();

   /* отображаем шар */
   glMaterialfv (GL_FRONT,GL_AMBIENT,mat3_amb);
   glMaterialfv (GL_FRONT,GL_DIFFUSE,mat3_dif);
```

```
   glMaterialfv (GL_FRONT,GL_SPECULAR,mat3_spec);
   glMaterialf  (GL_FRONT,GL_SHININESS,mat3_shininess);

   glPushMatrix ();
   glTranslatef (0.75, 0.0, -1.0);
   glutSolidSphere (1.0, 15, 15);
   glPopMatrix ();

   glPopMatrix ();
   /* выводим сцену на экран */
   glFlush ();
}

/* Вызывается при изменении пользователем размеров окна
 */
void reshape(int w, int h)
{
  /* устанавливаем размер области вывода
   * равным размеру окна
   */
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);

  /* задаем матрицу проекции с учетом размеров окна */
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();

  gluPerspective(
    40.0, /* угол зрения в градусах */
    (GLfloat)w/h, /* коэффициент сжатия окна */
    1,100.0);  /* расстояние до плоскостей отсечения */
  glMatrixMode (GL_MODELVIEW);

  glLoadIdentity ();
  gluLookAt(
   0.0f,0.0f,8.0f, /* положение камеры */
   0.0f,0.0f,0.0f, /* центр сцены */
   0.0f,1.0f,0.0f); /* положительное направление оси y */
}

/* Вызывается при нажатии клавиши на клавиатуре */
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27: /* escape */
         exit(0);
         break;
   }
```

```
}

/* Главный цикл приложения.
 * Создается окно, устанавливается режим
 * экрана с буфером глубины
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                        | GLUT_DEPTH);
   glutInitWindowSize (500, 500);
   glutCreateWindow (argv[0]);
   init ();
   glutReshapeFunc (reshape);
   glutDisplayFunc (display);
   glutKeyboardFunc (keyboard);
   glutMainLoop();
   return 0;
}
```

D.3. Загрузка BMP файла

В этом пункте приводится исходный текст функции *LoadBMP()*, которая позволяет загружать файлы полноцветных изображений (24 бита на точку) в формате Windows Bitmap (BMP).

Синтаксис вызова функции:

```
int LoadBMP(const char* filename, IMAGE* out_img)
```

Параметр *filename* определяет имя файла. Результат выполнения функции записывается в структуру *out_img,* которая определена следующим образом:

```
typedef struct _IMAGE
{
  int width;
  int height;
  unsigned char* data;
} IMAGE;
```

Поля *width* и *height* хранят, соответственно, высоту и ширину изображения. В поле *data* построчно хранится само изображение, в виде последовательности RGB-компонент цветов пикселей.

Файл **loadbmp.h:**

```
#ifndef _LOADBMP_H
#define _LOADBMP_H

typedef struct _IMAGE
  {
  int width;
  int height;
  unsigned char* data;
  } IMAGE;

int LoadBMP(const char* file, IMAGE* out_img);

#endif
```

Файл **loadbmp.c:**

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

#include "loadbmp.h"

/* размер заголовка BMP-файла */

#define BMP_SIZE_FILEHEADER 14

/* размер информационного заголовка BMP-файла */
#define BMP_SIZE_INFOHEADER 40

#define BMP_COLOR_BITS_24 24

/* вспомогательные функции */
/* <<<<<<<<<<<<<<<<<<<<<<< */

static unsigned int uInt16Number(unsigned char buf[2])
{
  return (buf[1] << 8) | buf[0];
}

static unsigned int uInt32Number(unsigned char buf[4])
{
  unsigned numb = buf[3];
  numb = (numb << 8) | buf[2];
  numb = (numb << 8) | buf[1];
  numb = (numb << 8) | buf[0];
```

```
  return numb;
}

int ReadFileHeader(FILE* f, int* bitmap_pos)
{
  unsigned char header[BMP_SIZE_FILEHEADER];
  int numb = 0;
  int offset = 0;

  if (fseek(f, 0, SEEK_SET))
    return 0;

  numb = fread(header, BMP_SIZE_FILEHEADER, 1, f);
  if (numb != 1)
    return 0;

  if (header[0] != 'B' || header[1] != 'M')
    return 0;

  offset = uInt32Number(header + 10);

  numb = fread(header, 4, 1, f);
  if (numb != 1)
    return 0;

  if (uInt32Number(header) != 40)
    return 0;

  *bitmap_pos  = offset;
  return 1;
}

/* <<<<<<<<<<<<<<<<<<<<<<< */

/* загрузка BMP-файла */

int LoadBMP(const char* file, IMAGE* out_img)
{
  FILE* f;
  int bitmap_pos;
  unsigned char buf[40];
  int numb;
  int x_res;
  int y_res;
  int n_bits;
  int compression;
  int size_image;
```

```
int n_used_colors;

/* открываем файл */
f = fopen(file, "rb");

if (!f) return 0;

if (out_img == NULL) return 0;

/* читаем заголовок */

if (!ReadFileHeader(f, &bitmap_pos))
{
  fclose(f);
  return 0;
}

if (fseek(f, BMP_SIZE_FILEHEADER, SEEK_SET))
{
  fclose(f);
  return 0;
}

numb = fread(buf, 40, 1, f);
if (numb != 1)
{
  fclose(f);
  return 0;
}

x_res = (int)uInt32Number(buf + 4);
y_res = (int)uInt32Number(buf + 8);

n_bits          = (int)uInt16Number(buf + 14);
compression     = (int)uInt32Number(buf + 16);
size_image      = (int)uInt32Number(buf + 20);
n_used_colors   = (int)uInt32Number(buf + 32);

/* читаем только полноцветные файлы */
if (n_bits == BMP_COLOR_BITS_24)
{
  int rgb_size;
  unsigned char* rgb;
  int y;
  unsigned char* line;
  int rest_4;
```

```
    if (bitmap_pos != BMP_SIZE_FILEHEADER +
BMP_SIZE_INFOHEADER)
    {
      fclose(f);
      return 0;
    }

    if (fseek(f, bitmap_pos, SEEK_SET))
    {
      fclose(f);
      return 0;
    }

    rgb_size = 3 * x_res;
    rest_4 = rgb_size % 4;
    if (rest_4 > 0)
      rgb_size += 4 - rest_4;

    out_img->width = x_res;
    out_img->height = y_res;

    out_img->data = (char*)malloc(x_res * y_res * 3);

    if (out_img->data == NULL)
      return 0;

    rgb = (unsigned char*)malloc(rgb_size);

    /* заполняем данные из файла */
    for (y = 0; y < y_res; y++)
    {
      int numb = 0;
      int x = 0;

      numb = fread(rgb, rgb_size, 1, f);
      if (numb != 1)
      {
        fclose(f);
        free(rgb);
        return 0;
      }

      numb = 0;
      line = out_img->data + x_res * 3 * y;
      for (x = 0; x < x_res; x++)
      {
        line[2] = rgb[numb++];
```

```
        line[1] = rgb[numb++];
        line[0]= rgb[numb++];
        line += 3;
      }
    }
    fclose(f);
    free(rgb);
  }
  else
    return 0;

  return 1;
}
```

D.4. Пример 3: Текстурирование

Результатом выполнения этой программы является построение тетраэдра с вращающимися вокруг него кольцами, на которые нанесена текстура.

При компиляции программы в MS Visual C++ файл ‘texture.bmp’ надо поместить в каталог проекта или указать полный путь к нему, используя символ ‘/’. Если путь не указан, то при запуске исполняемого файла из операционной системы, файл с текстурой должен находиться в том же каталоге. Для загрузки изображения текстуры программа использует функцию LoadBMP, приведенную в предыдущем пункте.

```
#include <GL\glut.h>
#include <gl\glaux.h>
#include <math.h>

#define TETR_LIST 1

GLfloat light_col[] = {1,1,1};
float mat_diff1[]={0.8,0.8,0.8};
float mat_diff2[]={0.0,0.0,0.9};
float mat_amb[]= {0.2,0.2,0.2};
float mat_spec[]={0.6,0.6,0.6};
float shininess=0.7*128,
float CurAng=0, RingRad=1, RingHeight=0.1;
GLUquadricObj* QuadrObj;
GLuint TexId;
GLfloat TetrVertex[4][3], TetrNormal[4][3];
```

```
/* Вычисление нормали к плоскости,
 * задаваемой точками a,b,c
 */
void getnorm (float a[3],float b[3],float c[3],float *n)
{
 float mult=0,sqr;
 int i,j;
 n[0]=(b[1]-a[1])*(c[2]-a[2])-(b[2]-a[2])*(c[1]-a[1]);
 n[1]=(c[0]-a[0])*(b[2]-a[2])-(b[0]-a[0])*(c[2]-a[2]);
 n[2]=(b[0]-a[0])*(c[1]-a[1])-(c[0]-a[0])*(b[1]-a[1]);
 /* Определение нужного направления нормали: от точки
(0,0,0) */
 for (i=0;i<3;i++)  mult+=a[i]*n[i];
 if (mult<0) for (j=0;j<3;j++) n[j]=-n[j];
}

/* Вычисление координат вершин тетраэдра */
void InitVertexTetr()
{
 float alpha=0;
 int i;
 TetrVertex[0][0]=0;
 TetrVertex[0][1]=1.3;
 TetrVertex[0][2]=0;
 /* Вычисление координат основания тетраэдра */
 for (i=1;i<4;i++)
 {
  TetrVertex[i][0]=0.94*cos(alpha);
  TetrVertex[i][1]=0;
  TetrVertex[i][2]=0.94*sin(alpha);
  alpha+=120.0*3.14/180.0;
 }
}

/* Вычисление нормалей сторон тетраэдра */
void InitNormsTetr()
{
 getnorm(TetrVertex[0], TetrVertex[1],
         TetrVertex[2], TetrNormal[0]);
 getnorm(TetrVertex[0], TetrVertex[2],
         TetrVertex[3], TetrNormal[1]);
 getnorm(TetrVertex[0], TetrVertex[3],
         TetrVertex[1], TetrNormal[2]);
 getnorm(TetrVertex[1], TetrVertex[2],
         TetrVertex[3], TetrNormal[3]);
}
```

```
/* Создание списка построения тетраэдра */
void MakeTetrList()
{
       int i;
  glNewList(TETR_LIST,GL_COMPILE);
  /* Задание сторон тетраэдра */
  glBegin(GL_TRIANGLES);
  for (i=1;i<4;i++)
  {
   glNormal3fv(TetrNormal[i-1]);
   glVertex3fv(TetrVertex[0]);
   glVertex3fv(TetrVertex[i]);
   if (i!=3) glVertex3fv(TetrVertex[i+1]);
     else glVertex3fv(TetrVertex[1]);
  }
  glNormal3fv(TetrNormal[3]);
  glVertex3fv(TetrVertex[1]);
  glVertex3fv(TetrVertex[2]);
  glVertex3fv(TetrVertex[3]);

  glEnd();
  glEndList();
}

void DrawRing()
{
 /* Построение цилиндра (кольца), расположенного
  * параллельно оси z. Второй и третий параметры
  * задают радиусы оснований, четвертый высоту,
  * последние два-число разбиений вокруг и вдоль
  * оси z. При этом дальнее основание цилиндра
  * находится в плоскости z=0
  */
  gluCylinder (QuadrObj,RingRad,RingRad,RingHeight,30,2);
}

void TextureInit()
{
 char strFile[]="texture.bmp";
 IMAGE img;

 /* Выравнивание в *.bmp по байту */
 glPixelStorei(GL_UNPACK_ALIGNMENT,1);
 /* Создание идентификатора для текстуры */
 glGenTextures(1,&TexId);
 /* Загрузка изображения в память */
 if (!LoadBMP(strFile, &img))
```

```
   return;

 /* Начало описания свойств текстуры */
 glBindTexture (GL_TEXTURE_2D,TexId);
 /* Создание уровней детализации и инициализация текстуры
  */
  gluBuild2DMipmaps(GL_TEXTURE_2D,3,img.width,
img.height,GL_RGB,GL_UNSIGNED_BYTE,img.data);

 /* Разрешение наложения этой текстуры на quadric-объекты
  */
 gluQuadricTexture (QuadrObj, GL_TRUE);
 /* Задание параметров текстуры */
 /* Повтор изображения по параметрическим осям s и t */
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);
 /* Не использовать интерполяцию при выборе точки на
  * текстуре
  */
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
 glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
 /* Совмещать текстуру и материал объекта */
 glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
}


void Init(void)
{
 InitVertexTetr();
 InitNormsTetr();
 MakeTetrList();
 /* Определение свойств материала */
 glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT, mat_amb);
 glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, mat_spec);
 glMaterialf  (GL_FRONT, GL_SHININESS, shininess);
 /* Определение свойств освещения */
 glLightfv(GL_LIGHT0, GL_DIFFUSE, light_col);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 /* Проводить удаление невидимых линий и поверхностей */
 glEnable(GL_DEPTH_TEST);
 /* Проводить нормирование нормалей */
```

```
 glEnable(GL_NORMALIZE);
 /* Материалы объектов отличаются только цветом диффуз-
  * ного отражения
  */
 glEnable(GL_COLOR_MATERIAL);
 glColorMaterial(GL_FRONT_AND_BACK,GL_DIFFUSE);
 /* Создания указателя на quadric-объект для построения
  * колец
  */
 QuadrObj=gluNewQuadric();
 /* Определение свойств текстуры */
 TextureInit();
 /* Задание перспективной проекции */
 glMatrixMode(GL_PROJECTION);
 gluPerspective(89.0,1.0,0.5,100.0);
 /* Далее будет проводиться только
  * преобразование объектов сцены
  */
 glMatrixMode(GL_MODELVIEW);
}

void DrawFigures(void)
{
 /* Включение режима нанесения текстуры */
 glEnable(GL_TEXTURE_2D);
 /* Задаем цвет диффузного отражения для колец */
 glColor3fv(mat_diff1);
 /* Чтобы не проводить перемножение с предыдущей матрицей
  * загружаем единичную матрицу
  */
 glLoadIdentity();
 /* Определяем точку наблюдения */
 gluLookAt(0.0, 0.0, 2.5,0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
 /* Сохраняем видовую матрицу, так как дальше будет
  * проводиться поворот колец
  */
 glPushMatrix();
  /* Производим несколько поворотов на новый
   * угол (это быстрее, чем  умножать предыдущую
   * видовую матрицу на матрицу поворота с фиксированным
   * углом поворота)
   */
  glRotatef (-CurAng,1,1,0);
  glRotatef (CurAng,1,0,0);
  /* Для рисования колец каждое из них надо
   * преобразовать отдельно, поэтому сначала сохраняем
   * видовую матрицу, затем восстанавливаем
```

```
   */
  glPushMatrix();
   glTranslatef (0,0,-RingHeight/2);
   DrawRing();
  glPopMatrix();
  glPushMatrix();
   glTranslatef (0,RingHeight/2,0);
   glRotatef (90,1,0,0);
   DrawRing();
  glPopMatrix();
  glPushMatrix();
   glTranslatef (-RingHeight/2,0,0);
   glRotatef (90,0,1,0);
   DrawRing();
  glPopMatrix();
 /* Восстанавливаем матрицу для поворотов тераэдра */
 glPopMatrix();
 /* Выключаем режим наложения текстуры */
 glDisable(GL_TEXTURE_2D);
 /* Проводим повороты */
 glRotatef (CurAng,1,0,0);
 glRotatef (CurAng/2,1,0,1);
 /* Чтобы тетраэдр вращался вокруг центра, его
  * надо сдвинуть вниз по  оси oz
  */
 glTranslatef (0,-0.33,0);
 /* Задаем цвет диффузного отражения для тетраэдра */
 glColor3fv(mat_diff2);
 /* Проводим построение тетраэдра */
 glCallList(TETR_LIST);
}

void Display(void)
{
 /* Инициализация (очистка) текущего буфера кадра и
  * глубины
  */
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 /* Построение объектов */
 DrawFigures();
 /* Перестановка буферов кадра */
 glutSwapBuffers();
}

void Redraw(void)
{
 /* Увеличение текущего угла поворота */
```

```
 CurAng+=1;
 /* Сигнал для вызова процедуры создания
  * изображения (для обновления)
  */
 glutPostRedisplay();
}

int main(int argc, char **argv)
{
  /* Инициализация функций библиотеки GLUT */
  glutInit(&argc, argv);
  /* Задание режима с двойной буферизацией,
   * представление цвета в  формате RGB, использование
   * буфера глубины
   */
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
  /* Создание окна приложения */
  glutCreateWindow("Example of using OpenGL");
  /* Регистрация функции построения изображения */
  glutDisplayFunc(Display);
  /* Регистрация функции обновления изображения */
  glutIdleFunc(Redraw);
  /* Инициализация функций OpenGL */
  Init();
  /* Цикл обработки событий */
  glutMainLoop();
  return 0;
}
```

Приложение E.

Примеры практических заданий

E.1. Cornell Box

Целью задания является создание изображения заданной трехмерной статичной сцены средствами OpenGL с использованием, возможно, стандартных геометрических примитивов.

Требуется создать изображение сцены Cornell Box. Эта классическая сцена представляет собой комнату кубического вида, с отсутствующей передней стенкой. В комнате находятся геометрические предметы различных форм и свойств (кубы, параллелепипеды, шары), и протяженный источник света на потолке. Присутствует также камера с заданными параметрами (обычно она расположена так, чтобы была видна вся комната). В одной из лабораторий Корнельского университета (http://graphics.cornell.edu) такая комната существует в реальности, и ее фотографии сравниваются с изображениями, построенными методами трассировки лучей для оценки точности методов. На странице лаборатории можно найти описание геометрии сцены в текстовом формате.

Реализации сцены, приведенной на рисунке достаточно для выполнения задания, хотя возможно введение новых предметов дополнительно к существующим, или вместо них. Приветствуется использование примитивов библиотек GLUT и GLU. Внимание! Сцена не должна превращаться в набор разнородных предметов. Эстетичность и оригинальность выполненного задания принимается во внимание.

Протяженный источник света на потолке комнаты можно эмулировать несколькими точечными источниками.

За простейшую реализацию сцены ставится 7 баллов.

Реалистичность сцены можно значительно повысить за счет разбиение многоугольников. Суть этого в том, что модели освещение OpenGL освещенность вычисляется в вершинах многоугольника с учетом направления нормалей в этих вершинах, а затем линейно интерполируется по всей поверхности. Если используются

относительно большие многоугольники, то, очевидно, невозможно получить действительно плавные переходы и затенения. Для преодоления этого недостатка можно разбивать большие грани (стены, например) на множество меньших по размерам. Соответственно разброс в направлении нормалей в вершинах одного многоугольника не будет столь велик и затенение станет более плавным. (1 балл)

Наложение текстур на объекты сцены поощряется 2-мя баллами.

Дополнительными баллами оценивается присутствие в сцене теней. Один из простейших алгоритмов наложения теней приведен в разделе 7.2. За его реализацию можно получить до 2 баллов. Использование более продвинутых алгоритмов (например, shadow volumes) будет оценено дополнительными баллами.

Реализация устранения ступенчатости (antialiasing) методом, предложенным в разделе 7.1. или каким-либо другим оценивается в 2 балла.

За введение в сцену прозрачных объектов и корректный их вывод дается 1 балл. Механизм работы с прозрачными объектами описан в разделе 6.1.

Задание оценивается, исходя из 15 баллов.

В приведенной ниже таблице указано распределение баллов в зависимости от реализованных требований:

Простейший вариант сцены (только освещение)	7 баллов
Разбиение полигонов	+1 балл
Использование текстур	+2 балла
Наложение теней	+2 балла
Устранение ступенчатости	+2 балла
Использование прозрачных объектов	+1 балл

Дополнительные баллы можно получить за хорошую оптимизацию программы, необычные решения, эстетичность и т.д.

E.2. Виртуальные часы

Целью задания является создание трехмерной интерактивной модели аналоговых часов.

Обязательные требования к программе:

1. Программа должна демонстрировать на экране трехмерную модель часов. Часы могут быть любые, от наручных до кремлевских. Проявите в полной мере Вашу фантазию и чувство меры! Постарайтесь сделать как можно более реалистичную сцену. Поощряется подробная детализация элементов часов.
2. Часы на экране обязательно должны иметь минутную и часовую стрелки. Секундная - по желанию, но очень приветствуется (иначе трудно будет определить, ходят часы или нет).
3. Время на часах должно совпадать с системным временем компьютера. Часы обязательно должны ходить, т.е. стрелки должны двигаться и скорость их движения не должна зависеть от производительности компьютера, а определяться только текущим временем.
4. Сцена должна быть интерактивной, т.е. давать приемлемую частоту кадров в секунду (>10) при визуализации на машине с аппаратным ускорителем трехмерной графики. Если программа будет работать медленно, баллы могут быть снижены
5. Необходимо реализовать вращения часов (или, возможно, камеры) с помощью мыши (предпочтительно) или клавиатуры. Можно также предусмотреть режимы с автоматическим вращением.

Пожелания к программе:

1. Поощряется введение дополнительной геометрии. Например, ремешков, маятников и т.д. Можно сделать часы с кукушкой, будильник и т.п.
2. Желательно наличие возможностей для управления процессом визуализации. Например, наличие/отсутствие текстур, режимы заливки, детализации и т.д.

3. Приветствуется выполнение задания в виде демонстрации, т.е. с возможностью работы в полноэкранном режиме и немедленным выходом по клавише Escape. Можно написать программу как Screen Saver.

4. Постарайтесь использовать максимум возможностей OpenGL. Блики, отражения, спецэффекты - за все это обязательно даются дополнительные баллы.

5. Проявите вкус - сделайте так, чтобы нравилось прежде всего Вам. Но не увлекайтесь - оставайтесь реалистами.

Максимальная оценка - 20 баллов. За минимальную реализацию требований ставиться 10 баллов. Еще до 10 баллов можно получить за использование в работе возможностей OpenGL (текстур, прозрачности , environment mapping и пр.), оригинальных и продвинутых алгоритмов, количество настроек, а также за эстетичность и красоту сцены.

E.3. Интерактивный ландшафт

Целью данного задания является генерация и вывод с помощью OpenGL поверхности ландшафта, а также обеспечение интерактивного передвижения над ней.

Обязательная часть задания

Для выполнения обязательной части задания необходимы:

1. генерация трехмерного ландшафта
2. раскраска для придания реалистичности
3. эффект тумана
4. возможность "полета" над ландшафтом (управление)

Более подробное описание:

Генерация ландшафта

Один из вариантов задания поверхности ландшафта - задание так называемого "поля высот" - функции вида z=f(x, y), которая сопоставляет каждой точке (x, y) плоскости OXY число z - высоту поверхности ландшафта в этой точке. Один из способов задания функции f - табличный, когда функция f представляется матрицей T размера M x N, и для целых x и y f=T[x, y], а для дробных x и y из диапазонов [0..M-1] и [0..N-1] соответственно f вычисляется интерполяцией значений f в ближайших точках плоскости OXY с

целыми x и y, а вне указанных диапазонов x и y значение функции считается неопределенным.

Допустим, в памяти лежит двухмерный массив со значениями матрицы T. Пусть N=M. Если теперь для каждого квадрата [x, x+1] x [y, y+1], где x и y принадлежат диапазону [0..N-2] построить две грани: ((x, y, T[x, y]), (x+1, y, T[x+1, y]), (x+1, y+1, T[x+1, y+1])) и ((x, y, T[x, y]), (x+1, y+1, T[x+1, y+1]), (x, y+1, T[x, y+1])), то мы получим трехмерную модель поверхности, описываемой матрицей T.

Но каким образом задать массив значений матрицы T? Один из способов - сгенерировать псевдослучайную поверхность с помощью фрактального разбиения. Для этого положим размерность матрицы T равной 2^N+1, где N - натуральное число. Зададим некоторые произвольные (псевдослучайные) значения для четырех угловых элементов матрицы T. Теперь для каждого из четырех ребер матрицы T (это столбцы или строки элементов, соединяющие угловые элементы) вычислим значение элемента матрицы T, соответствующего середине ребра. Для этого возьмем среднее арифметическое значений элементов матрицы T в вершинах ребра и прибавим к получившемуся значению некоторое псевдослучайное число, пропорциональное длине ребра. Значение центрального элемента матрицы T вычислим аналогично, только будем брать среднее арифметическое четырех значений элементов матрицы в серединах ее ребер.

Теперь разобьем матрицу T на четыре квадратные подматрицы. Значения их угловых элементов уже определены и мы можем рекурсивно применить к подматрицам T описанную выше процедуру. Будем спускаться рекурсивно по дереву подматриц, пока все элементы T не будут определены. С помощью подбора коэффициентов генерации псевдослучайной добавки можно регулировать "изрезанность" поверхности. Для реалистичности поверхности важно сделать величину псевдослучайной добавки зависящей от длины текущего ребра - с уменьшением размера ребра должно уменьшаться и возможное отклонение высоты его середины от среднего арифметического высот его вершин.

Один из других вариантов - использовать изображения в градациях серого для карты высот. (В этом случае ландшафт можно оттекстурировать с помощью соответствующей цветной картинки и линейной генерации текстурных координат)

Внимание: использование NURBS возможно, но не приветствуется в силу ограниченности использования NURBS для реальных приложений.

Раскраска ландшафта

Чтобы сделать получившуюся модель немного более напоминающей ландшафт, ее можно раскрасить. Каждой вершине можно сопоставить свой цвет, зависящий от высоты этой вершины. Например, вершины выше определенного уровня можно покрасить в белый цвет в попытке сымитировать шапки гор, вершины пониже - в коричневый цвет скал, а вершины уровнем еще ниже - в зеленый цвет травы. Значения "уровней" раскраски поверхности следует подобрать из эстетических соображений.

Освещение ландшафта

Для еще большего реализма и для подчеркивания рельефа осветить модель ландшафта бесконечно удаленным источником света (как бы солнцем).

Цвет вершин можно задавать через `glColor*()` совместно с `glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);`

Туман

Чтобы усилить (или хотя бы создать) иллюзию больших размеров модели и ее протяженности, можно воспользоваться эффектом тумана. Тип тумана (линейный или экспоненциальный) следует выбрать из индивидуальных эстетических предпочтений. Способ создания тумана описан в разделе 4.4.

Управление

Элементарное управление движением камеры по клавиатурным "стрелочкам". Нажатие на стрелку "вверх" - передвижение по направлению взгляда вперед. "Назад" - по направлению взгляда назад. "Влево", "Вправо" по аналогии, "Page Up", "Page Down" - вверх, вниз, соответственно.

В GLUT'е получать нажатия не алфавитно-цифровых клавиш можно через функцию `glutSpecialFunc(void (*)(int key, int x, int y))`, где key - константа, обозначающая клавишу (см. в glut.h - **GLUT_KEY_**). Функция используется аналогично `glutKeyboardFunc()`.

Дополнительная часть

Управление мышью:

Движение мыши в горизонтальной плоскости (смещение по оси X) управляет углом поворота направления взгляда в горизонтальной плоскости (альфа, от 0 до 2*PI). Движение мыши в вертикальной плоскости (смещение по оси Y) управляет углом поворота направления взгляда в вертикальной плоскости относительно горизонта (бета, от -PI до PI). Зная оба угла, вектор направления взгляда в мировых координатах вычисляется следующим образом:

direction_z = sin(бета);
direction_x = cos(альфа) * cos(бета);
direction_y = sin(альфа) * cos(бета),

а затем нормализуется.

Вектор направления "вбок" вычисляется как векторное произведение вектора направления вертикально вверх, то есть вектора (0, 0, 1) и уже известного вектора направления взгляда.

Вектор направления "вверх" вычисляется как векторное произведение вектора направления взгляда и вектора направления "вбок".

Положение камеры в OpenGL можно передать через `gluLookAt()`. Подсказка: параметр *target* можно положить равным *position* + *direction*

Смещение позиции камеры должно происходить не на фиксированное расстояние за один кадр, а вычисляться, исходя из скорости передвижения камеры, и времени, ушедшего на обсчет последнего кадра. Передвижение камеры должно осуществляться в направлении взгляда. Скажем, по левой кнопке мыши - вперед, а по правой - назад. Для того, чтобы засечь время, можно воспользоваться функцией `timeGetTime()`, описанной в "mmsystem.h", и реализованной в библиотеке "winmm.lib" (только для Windows)

```
#include "mmsystem.h"
...
void Display()
{
...
int system_time_before_rendering;
system_time_before_rendering = timeGetTime();
RenderFrame();
int time_spent_rendering_msec =
        timeGetTime() -system_time_before_rendering;
...
}
```

В GLUT'е для этого есть специальный вызов
`time = glutGet(GLUT_ELAPSED_TIME)` (аналогично `timeGetTime()`)

Вода, или нечто на нее похожее

При раскраске ландшафта можно добавить еще один, самый нижний "уровень" - уровень воды. Вершины, располагающиеся на этом уровне можно покрасить в цвет воды - предположительно, синий. Для того, чтобы получившиеся "водоемы" не выглядели продолжением поверхности ландшафта просто покрашенным в синий цвет, а имели плоскую поверхность, при генерации поля высот можно установить порог высоты, ниже которого "опускаться" вершинам запрещается. Если для элемента матрицы генерируется значение высоты ниже этого порога, элемент инициализируется пороговым значением.

Объекты

По ландшафту можно раскидать в художественном беспорядке от пятидесяти (50) объектов, встречающихся на ландшафте в обычной жизни, например домов или деревьев. При этом ель считается деревом, а две равнобедренные вытянутые вертикально грани, поставленные на ландшафт крест накрест и покрашенные в зеленый цвет, считаются елью.

Отражения в воде

Сделать так, чтобы ландшафт отражался в воде, которая уже должна присутствовать на ландшафте (то есть подразумевается, что это дополнительное задание является развитием дополнительного задания 2). Один из вариантов реализации: рассчитав текущую матрицу камеры, отразить ее относительно плоскости воды и изображение ландшафта, не выводя при этом грани поверхности воды. Затем, пользуясь еще не отраженной матрицей камеры, визуализировать грани поверхности воды полупрозрачными. Это создаст иллюзию поверхности воды, сквозь которую видно отражение. Затем, опять же с неотраженной матрицей камеры, нужно нарисовать сам ландшафт. (этот подход является частным случаем описанного в разделе 7.3.)

Тени

На этапе раскраски вершин ландшафта (то есть это надо сделать один раз, а не каждый кадр) из каждой вершины можно выпустить луч, противоположный направлению солнца. Если луч не пересекся с поверхностью ландшафта - раскрашивать как запланировано, если пересекся - значит данная вершина ландшафта находится в тени и для

нее нужно взять менее интенсивный цвет. Примечание: реализация теней является задачей повышенной сложности (придется писать нахождение пересечений луча с гранями, что в общем случае нетривиально).

Оценка:

База

Ландшафт	8 баллов
Раскраска	2 балла
Управление	2 балла

Дополнительно

Управление мышью	+2 балла
Объекты	+3 балла
Вода	+4 балла
Отражение	+4 балла
*Тени	+5 баллов
Всего	30 баллов

В таблице указано максимальное число баллов по каждому пункту. Система выставления баллов - гибкая, зависит от правдоподобности и впечатления от работы.

Дополнительные источники информации:

http://www.vterrain.org

Литература

1. Каннингем, С. ACM SIGGRAPH и обучение машинной графике в Соединенных штатах. Программирование, 4, 1991.
2. Bayakovsky, Yu. Russia: Computer Graphics Education Takes off in 1990s. Computer Graphics, 30(3), Aug. 1996.
3. Canningham S. An Evoluing Approach to CG Courses in CS. Graphicon'98 Conference Proceedings, MSU, Sept. 1998.
4. Bayakovsky, Yu. Virtual Laboratory for Computer Graphics and Machine Vision. Graphicon'99, Conference proceedings, MSU, Sept 1999.
5. Эйнджел Э. Интерактивная компьютерная графика. Вводный курс на базе OpenGL, 2 изд. Пер. с англ.- Москва, «Вильямс», 2001.
6. Порев В.Н. Компьютерная графика. СПб., BHV, 2002.
7. Шикин А. В., Боресков А. В. Компьютерная графика. Полигональные модели. Москва, ДИАЛОГ-МИФИ, 2001.
8. Тихомиров Ю. Программирование трехмерной графики. СПб, BHV, 1998.
9. Performance OpenGL: Platform Independent Techniques. SIGGRAPH 2001 course.
10. OpenGL performance optimization, Siggraph'97 course.
11. Visual Introduction in OpenGL, SIGGRAPH'98.
12. The OpenGL graphics system: a specification (version 1.1).
13. Программирование GLUT: окна и анимация. Miguel Angel Sepulveda, LinuxFocus.
14. The OpenGL Utility Toolkit (GLUT) Programming Interface, API version 3, specification.

Предметный указатель

A

API 8

G

GLU, Graphics Utility Library . 11
GLUT, GL Utility Toolkit 11

I

IRIS GL 8

O

OpenGL 8
  оптимизация 76
  ошибки 87
  приемы работы 66
  синтаксис команд 14

Б

Буфер
  глубины 39, 58
  кадра 58, 59, 62
  маски 58, 62, 71, 73
  накопитель 58, 61
  очистка 20, 86
  цвета 58
Буферизация
  двойная 58

В

Вершина 12, 21
  атрибуты 12, 30
  массив 29
  нормаль 21, 22
  положение 21
  цвет 21, 22

Г

Грань 26
  лицевая 26
  обратная 27

Д

Дисплейный список 28, 82
  вызов 28
  создание 28
  удаление 28

З

Зеркальные отражения 72

И

Источник света 43
  добавление 43

К

Команды GL
  glAccum 61
  glArrayElement 30
  glBegin 23
  glBindTexture 51
  glBlendFunc 59
  glCallList 28
  glCallLists 28
  glClear 20, 21
  glClearColor 20
  glClearDepth 87
  glColor 22
  glColorMaterial 42
  glColorPointer 29
  glCullFace 27
  glDeleteLists 28

glDepthRange 39
glDisable 23
glDisableClientState 30
glDrawArrays 30
glDrawBuffer 61
glDrawElements 30
glEnable 23
glEnableClientState 30
glEnd 23
glEndList 28
glFog 47
glFrontFace 26
glGenTextures 51
glHint 64
glLight 43, 106
glLightModel 40
glLoadIdentity 33
glLoadMatrix 33
glMaterial 41
glMatrixMode 33
glMultMatrix 34
glNewList 28
glNormal 22
glNormalPointer 29
glOrtho 36
glPointSize 78
glPolygonMode 26
glPopMatrix 33
glPushMatrix 33
glReadBuffer 61
glRotate 35
glScale 35
glShadeModel 22
glStencilFunc 62
glStencilOp 62
glTexCoord 55
glTexEnv 54
glTexGen 55
glTexParameter 52
glTranslate 35
gluLookAt 36
glVertex 21
glVertexPointer 29
glViewPort 38
Команды GLAUX
auxDIBImageLoad 49
Команды GLU
gluBuild2DMipmaps 51
gluCylinder 27, 93
gluDisk 93
gluNewQuadric 27
gluOrtho2D 36
gluPartialDisk 94
gluPerspective 37
gluQuadricTexture 55
gluScaleImage 50
gluSphere 27, 93
Команды GLUT
glutCreateWindow 18
glutDisplayFunc 18, 20, 91
glutIdleFunc 91
glutInit 17, 89
glutInitDisplayMode 18, 90
glutInitWindowPosition 90
glutInitWindowSize 18, 90
glutKeyboardFunc 18
glutMainLoop 18, 91
glutMotionFunc 91
glutMouseFunc 91
glutPassiveMotionFunc 91
glutPostRedisplay 91
glutPostRedisplay 20
glutReshapeFunc 18, 39, 91
glutSolidCone 95
glutSolidCube 94
glutSolidDodecahedron 95
glutSolidIcosahedron 95
glutSolidOctahedron 95
glutSolidSphere 94
glutSolidTetrahedron 95
glutSolidTorus 95
glutWireCone 95

glutWireCube 94
glutWireDodecahedron 95
glutWireIcosahedron 95
glutWireOctahedron 95
glutWireSphere..... 94
glutWireTetrahedron 95
glutWireTorus 95
Конвейер OpenGL 12
режим работы 23
Конус видимости..... 37

Л

Лестничный эффект 66
устранение..... 66

М

Материал
параметры 41
Матрица..... 32
единичная..... 33
изменение..... 33
модельно-видовая..... 32
проекций..... 32
создание..... 35
сохранение 33
текстуры 32
текущая..... 35
умножение..... 34

О

Область вывода..... 38
Операторные скобки 23
Освещение
модель..... 40

П

Положение наблюдателя 35
Примитив 12, 58
атомарный..... *См.* Вершина
интерполяция цветов..... 22
многоугольник..... 24
отрезок..... 24
последовательность..... 23
связанный..... 80
тип..... 23
точка 24
треугольник..... 24
четырехугольник 24
Проекция 36
ортографическая..... 36
перспективная..... 37
Прозрачность 59

Р

Растеризация 58

С

Сервер OpenGL..... 12
Система координат..... 32
левосторонняя..... 36
оконная 38

Т

Текстура 49
координаты 55
наложение 52
подготовка..... 49
размеры 50
режим интерполяции..... 53
уровень детализации 50
Тени 67
Туман..... 46, 58
вычисление интенсивности47

Ф

Функция с обратным вызовом
..... 17
обновления изображения ... 20