37
Министерство образования и науки Украины
Луганский национальный педагогический университет имени Тараса Шевченко
Институт экономики и бизнеса
Курсовая работа
На тему:
Использование OpenGL

Выполнил
Студент 3 курса
Кравченко А.С.
Проверил
Кутепова Л.М.

Луганск 2004
Оглавление

Введение

Библиотека OpenGL представляет собой программный интерфейс для аппаратного обеспечения машинной графики. Этот интерфейс состоит приблизительно из 250 отдельных команд (почти 200 команд в ядре OpenGL и еще 50 команд в библиотеке утилит OpenGL), которые используются для того, чтобы определить объекты и операции, необходимые для создания интерактивных трехмерных прикладных программ.

Библиотека OpenGL разработана в качестве низкоуровневого, аппаратно-независимого интерфейса, допускающего реализацию на множестве различных аппаратных платформ. Для того чтобы достичь этих качеств, в состав библиотеки OpenGL не включены никакие команды для выполнения задач работы с окнами или для получения пользовательского ввода; вместо этого вы должны работать через любую систему управления окнами, которая работает с конкретными аппаратными средствами. Точно так же библиотека OpenGL не предоставляет команды высокого уровня для описания моделей трехмерных объектов. Такие команды могли бы позволить определять относительно сложные формы, например, автомобили, части тела, самолеты или молекулы. При использовании библиотеки OpenGL вы должны создавать нужную модель из ограниченного набора геометрических примитивов -- точек, линий и многоугольников.

Более сложная библиотека, которая обеспечивает эти функциональные возможности, конечно, могла бы быть создана поверх библиотеки OpenGL. Библиотека утилит OpenGL (GLU -- OpenGL Utility Library) предоставляет множество возможностей моделирования, таких как поверхности второго порядка и NURBS-кривых и поверхностей (NURBS -- Non-Uniform, Rational B-Spline -- неравномерный рациональный B-сплайн). Библиотека GLU представляет собой стандартную часть каждой реализации OpenGL. Существуют также наборы инструментов более высокого уровня, такие как FSG (Fahrenheit Scene Graph), которые являются надстройкой библиотеки OpenGL, и самостоятельно доступны для множества реализаций библиотеки OpenGL.

1.1. Программный код OpenGL

Поскольку с помощью графической системы OpenGL можно решить так много задач, OpenGL-программа может быть достаточно трудной для понимания. Однако основная структура полезной программы может быть проста: ее задачи состоят в том, чтобы инициализировать некоторые состояния, которые управляют тем, как библиотека OpenGL выполняет визуализацию, и определить объекты, которые будут визуализированы.

Прежде чем приступить к анализу некоторого программного кода OpenGL, давайте познакомимся с несколькими терминами. Визуализация, с ее использованием вы уже сталкивались, представляет собой процесс, посредством которого компьютер создает изображения из моделей. Эти модели, или объекты, создаются из геометрических примитивов, -- точек, линий и многоугольников, -- которые определяются их вершинами.

Конечное визуализированное изображение состоит из пикселей, выводимых на экран; пиксель представляет собой наименьший видимый элемент, который аппаратные средства отображения могут поместить на экран.

Информация о пикселях (например, какой цвет предполагается для этих пикселей) организована в памяти в виде битовых плоскостей. Битовая плоскость представляет собой область памяти, которая содержит один бит информации для каждого пикселя на экране; этот бит мог бы указывать, например, на то, что конкретный пиксель, как предполагается, является красным. Битовые плоскости, в свою очередь, организованы в буфер кадра, который содержит всю информацию, необходимую графическому дисплею для того, чтобы управлять цветом и яркостью всех пикселей на экране.

Пример 1. демонстрирует визуализацию белого прямоугольника на черном фоне, как это показано на рисунке 1.

Пример 1. Фрагмент программного кода OpenGL

```
finclude <whateverYouNeed.h> ,
main () {
InitializeAWindowPlease();
glClearColor(0.0, 0.0, 0.0, 0.0); glClear(GL_COLOR_BUFFER_BIT); glColor3f(1.0, 1.0, 1.0);
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0); glBegin(GL_POLYGON);
glVertex3f(0.25,  0.25,  0.0);
glVertex3f (0.75,  0.25,  0.0);
glVertex3f(0.75,  0.75,  0.0);
glVertex3f(0.25,  0.75,  0.0); glEnd() ; glFlush () ;
UpdateTheWindowAndCheckForEvents(); }
```

Первая строка функции main() инициализирует определенное окно на экране: функция InitializeAWindowPlease() используется в данном случае в качестве метки-"заполнителя" для подпрограмм специфических оконных систем, которые в общем случае не являются вызовами OpenGL. Следующие две строки содержат команды OpenGL, которые устанавливают черный цвет фона для окна: функция glCIearCoIor() определяет то, какой цвет фона будет установлен для окна, а функция glClear() фактически устанавливает цвет окна. Как только цвет фона установлен, окно

заливается этим цветом всякий раз, когда вызывается функция glClear(). Этот цвет фона может быть изменен с помощью второго вызова функции glClearColor(). Точно так же функция glColor3f() устанавливает то, какой цвет следует использовать для прорисовки объектов на экране -- в данном случае этот цвет является белым. Все объекты, выводимые на экран после этого момента, используют данный цвет до тех пор, пока он не будет изменен с помощью следующего вызова команды установки цвета.

Следующая функция OpenGL, используемая в рассматриваемой программе, glOrtho(), определяет систему координат, которую OpenGL принимает для прорисовки окончательного изображения, и то, как это изображение отображается на экране. Вызовы, заключенные между функциями glBegin() и glJnd(), определяют объект, который будет выведен на экран, в рассматриваемом примере это многоугольник с четырьмя вершинами. "Углы" многоугольника определяются с помощью функции glVertex3f(). Как вы, наверное, уже догадались, исходя из значений параметров этой функции, которые представляют собой координаты (x, y, z), данный многоугольник является прямоугольником, расположенным на плоскости z(z -- 0).

Наконец, функция glFlush() гарантирует, что команды прорисовки фактически выполняются, а не просто сохраняются в некотором буфере, ожидая дополнительных команд OpenGL. Подпрограмма-"заполнитель" UpdateTheWindowAndCheckForEvents() управляет содержимым окна и начинает обработку событий.

1.2. Синтаксис команд OpenGL

Как вы, вероятно, могли заметить из примера простой программы, приведенного в предшествующем разделе, команды библиотеки OpenGL используют префикс gl. Каждое слово, составляющее наименование команды, начинается с заглавной буквы (вспомните, например, функцию glClearColor()). Точно так же имена констант, определенных в библиотеке OpenGL, начинаются с префикса GL_, записываются целиком заглавными буквами и используют символы подчеркивания, чтобы разделить отдельные слова (например, GL__COLOR_BUFFER_BIT).

Вы, вероятно, также смогли заметить некоторые символы, которые показались вам посторонними, они добавляются в конец наименования некоторых команд (например, 3f в функциях glColor3f() и glVertex3f()). Действительно, часть Color в наименовании функции glColor3f() достаточна для того, чтобы определить данную команду как команду, устанавливающую текущий цвет. Однако были определены несколько таких команд, чтобы вы смогли использовать их с различными типами параметров. В частности, часть 3 суффикса указывает, что для этой команды задаются три параметра; другая версия команды Color использует четыре параметра. Часть f суффикса указывает на то, что параметры данной команды представляют собой числа с плавающей точкой. Наличие различных форматов позволяет библиотеке OpenGL принимать данные пользователя в его собственном формате данных.

Некоторые команды библиотеки OpenGL допускают использование 8 различных типов данных в качестве своих параметров. Буквы, используемые в качестве суффиксов для того, чтобы определить эти типы данных для реализации ISO C

библиотеки OpenGL, представлены в Таблице 1.1; там же приведены соответствующие определения типов в библиотеке OpenGL. Конкретная реализация библиотеки OpenGL, которую вы используете, может не совпадать в точности с приведенной схемой; например, реализации для языков программирования C++ или Ada, не требуют этого.

| Тип данных | Типичный соответствующий тип данных языка программирования C | Определение типов данных библиотеки OpenGL |
|---|---|---|
| 8-разрядное целое | signed char | GLbyte |
| 16-разрядное целое | | |

short

GLshort

32-разрядное целое

Int или long

GLint, GLsizei

32-разрядное число
с плавающей точкой

float

GLfloat, GLclampf

64-разрядное число
с плавающей точкой

double

GLdouble, GLclampd

8-разрядное беззнаковое целое

unsigned char

GLubyte, GLboolean

16-разрядное беззнаковое целое

unsigned short

GLushort

32-разрядное беззнаковое целое

unsignedint или
unsigned long

GLuint, GLenum, GLbitfield

Таблица 1.1 Суффиксы наименований команд и типы данных параметров

Таким образом, две команды
glVertex2i (1,3); glVertex2f (1.0, 3.0);
являются эквивалентными, за исключением того, что первая из них определяет координаты вершины как 32-разрядные целые числа, а вторая определяют их как числа с плавающей точкой с одинарной точностью.
Наименования некоторых команд библиотеки OpenGL могут иметь заключительный символ v, который указывает на то, что данная команда принимает указатель на вектор (или массив) значений, а не набор индивидуальных параметров. Много команд имеют как векторные, так и невекторные версии, но некоторые команды принимают только индивидуальные параметры, тогда как другие команды требуют, чтобы, по крайней мере, некоторые из их параметров были определены как векторы. Следующие строки показывают, как можно было бы использовать векторную и невекторную версию команды, которая устанавливает текущий цвет:
glColor3f (1.0, 0.0, 0.0);
gLfloat color_array [] = {1.0, 0.0, 0.0); glColor3fv (color_array);
Наконец, библиотека OpenGL определяет тип данных GLvoid. Этот тип данных наиболее часто используется для тех команд библиотеки OpenGL, которые принимают в качестве параметров указатели на массивы значений.
1.3 OpenGL как конечный автомат
Графическая система OpenGL представляет собой конечный автомат. Вы переводите этот автомат в различные состояния (или режимы), которые затем остаются в силе до тех пор, пока вы не измените их. Как уже было показано выше, текущий цвет представляет собой переменную состояния. Можно установить в качестве текущего белый, красный, или любой другой цвет, и после этого каждый объект будет выводиться на экран с этим цветом до тех пор, пока вы не установите для текущего цвета какое-нибудь другое значение. Текущий цвет является лишь одной из множества переменных состояния, которые поддерживает библиотека OpenGL. Другие переменные состояния управляют такими вещами, как текущая визуализация и преобразования проецирования, шаблоны штриховки линий и многоугольников, режимы вывода многоугольников на экран, соглашения по упаковке пикселей, местоположение и характеристики источников освещения, а также свойства материалов объектов, выводимых на экран. Множество переменных состояния относятся к режимам, которые включаются или отключаются с помощью команд glEnable() или glDisable() соответственно.
Каждая переменная состояния или режим имеют значение по умолчанию, и в любой момент времени можно запросить у системы текущие значения каждой из этих переменных. Как правило, для этого используется одна из шести команд, которые перечислены далее: glGetBooleanv(), glGetDoubIev(), glGetFloatv(), glGetlntegerv(), glGetPointerv() или gIIsEnabled(). То, какую из этих команд следует выбрать, зависит от того, в виде какого типа данных вы хотите получить ответ. Для некоторых переменных состояния используются более конкретные команды запроса (такие как glGetLight*(), glGetError() или glGetPolygonStipple()). Кроме этого, можно сохранить

набор переменных состояний в стеке атрибутов с помощью команд glPushAttrib() или gIPushCIientAttrib(), временно изменить их, а затем восстановить значения этих переменных с помощью команд glPopAttrib() или glPopClienlAttrib(). Для временных изменений состояния необходимо использовать именно эти команды, а не любые из команд запроса, так как они более эффективны.

## 1.4. Конвейер визуализации OpenGL

### 1.4.1. Конвейер

Большинство реализаций библиотеки OpenGL имеет одинаковый порядок операций, определенную последовательность стадий обработки, которая называется конвейером визуализации OpenGL. Этот порядок функционирования, показанный на рисунке 1.2, не является строгим правилом реализации библиотеки OpenGL, однако он представляет собой надежное руководство для предсказания результата работы OpenGL.

Следующая диаграмма демонстрирует концепцию сборочного конвейера Генри Форда, которую библиотека OpenGL использует для обработки данных. Геометрические данные (вершины, прямые и многоугольники) проходят через последовательность блоков, в число которых входят вычислители и операции обработки вершин, в то время как пиксельные данные (пиксели, изображения и растровые образы) для определенной части процесса визуализации обрабатываются по-другому. Оба типа данных подвергаются одним и тем же заключительным операциям (операции растеризации и пофрагментной обработки), прежде чем итоговые пиксельные данные записываются в буфер кадра.

Рис. 1.2 Конвейер визуализации

### 1.4.2 Списки вывода

Все данные, вне зависимости от того, описывают ли они геометрию или пиксели, могут быть сохранены в списке вывода для текущего или более позднего использования. (Альтернативой сохранению данных в списке вывода является немедленная обработка данных, она также известна под названием непосредственный режим работы.) После того как список вывода был создан, сохраненные данные посылаются из этого списка вывода точно так же, как если бы они были посланы прикладной программой в непосредственном режиме работы.

### 1.4.3 Вычислители

Все геометрические примитивы, в конечном счете, описываются вершинами. Параметрические кривые и поверхности могут быть первоначально описаны контрольными точками и полиномиальными функциями, которые называются базовыми функциями. Вычислители предоставляют метод получения вершин, используемый для представления поверхности по контрольным точкам. Таким методом является полиномиальное отображение, которое может формировать нормаль к поверхности, координаты текстуры, цвета и значения пространственных координат по контрольным точкам.

### 1.4.4 Операции обработки вершин

Для данных по вершинам следующей является стадия "операций обработки вершин", которая преобразовывает вершины в примитивы. Некоторые типы данных по

вершинам (например, пространственные координаты) преобразовываются в матрицы чисел с плавающей точкой размером 4x4. Пространственные координаты проецируются из положения в трехмерном пространстве в положение на вашем экране.

Если разрешено использование расширенных функциональных возможностей, то данная стадия обработки данных еще более насыщена. Если используется наложение текстур, то на этой стадии могут быть сгенерированы и преобразованы координаты текстуры. Если разрешено освещение, то здесь выполняются вычисления параметров освещения, для чего используются преобразованные вершины, нормаль к поверхности, положение источника освещения, свойства материала и другая информация освещения, необходимая для получения значения цвета.

1.4.5 Сборка примитивов

Операция отсечения, основная часть сборки примитивов, представляет собой удаление частей геометрии, которые выходят за пределы полупространства, определенного некоторой плоскостью. При отсечении точек просто пропускаются или отбрасываются вершины; при отсечении линий или многоугольников могут добавляться дополнительные вершины в зависимости от того, как отсекается линия или многоугольник.

В некоторых случаях этот процесс сопровождается перспективным делением, которое заставляет удаленные геометрические объекты казаться меньше, чем более близкие объекты. После этого применяются операции получения окна просмотра (Viewport) и глубины (г-координата). Если отбраковка разрешена, и данный примитив представляет собой многоугольник, тогда он может быть отброшен в процессе выполнения теста на отбраковку. В зависимости от способа построения многоугольник может быть выведен на экран в виде точек или в виде линий.

Результатом выполнения этой стадии являются законченные геометрические примитивы, которые представляют собой преобразованные и отсеченные вершины и связанные с ними значения цвета, глубины и иногда координаты текстур, а также указания для выполнения стадии растеризации.

1.4.6 Операции обработки пикселей

В то время как геометрические данные следуют одним путем по конвейеру визуализации OpenGL, пиксельные данные следуют другим маршрутом. Пиксели из определенного массива в системной памяти сначала распаковываются из какого-либо одного из множества форматов в надлежащее количество компонентов. Затем эти данные масштабируются, смещаются и обрабатываются посредством карты элементов отображения. После этого результаты фиксируются и либо записываются в область памяти, выделенную под текстуры, либо передаются на стадию растеризации. (См. раздел "Конвейер отображения" в Главе 8.)

Если пиксельные данные считываются из буфера кадра, то выполняются операции по передаче пикселя (масштабирование, смещение, отображение и фиксация). Затем полученные результаты упаковываются в соответствующий формат и возвращаются в некоторый массив системной памяти.

Существуют специальные операции копирования пикселей для копирования данных

из буфера кадра в другие части буфера кадра или в область памяти, выделенную для текстур. Выполняется однопроходная реализация операций при передаче пикселя, а затем данные записываются в область памяти, выделенную для текстур или обратно в буфер кадра.

1.4.7 Сборка текстуры

OpenGL-приложения могут накладывать изображения текстуры на геометрические объекты для того, чтобы сделать их вид более реалистичным. Если используется несколько изображений текстуры, то весьма разумно будет поместить их в объекты текстуры для упрощения переключения между ними.

Некоторые реализации библиотеки OpenGL могут иметь специальные ресурсы для ускоренного выполнения операций над текстурами. Это может быть реализовано как специализированная, высокопроизводительная область памяти, выделенная для текстур. Если такая память доступна, объекты текстуры могут быть упорядочены по приоритетам для облегчения управления этим ограниченным и ценным ресурсом.

1.4.8. Растеризация

Растеризация представляет собой преобразование как геометрических, так и пиксельных данных во фрагменты. Каждый квадратный фрагмент соответствует определенному пикселю в буфере кадра. Штриховка линий и многоугольников, ширина линии, размер точки, модель закраски и вычисления покрытия, необходимые для поддержки сглаживания, учитываются как вершины, которые соединяются в линии, или как внутренние пиксели, рассчитанные для закрашенного многоугольника. Значения цвета и глубины назначаются для каждого квадратного фрагмента.

1.4.9 Операции обработки фрагментов

Прежде чем значения фактически сохраняются в буфере кадра, выполняется ряд операций, в результате чего фрагменты могут быть изменены или даже отброшены. Все эти операции могут быть включены или отключены.

Первой операцией, с которой можно столкнуться, является наложение текстур. Эта операция заключается в том, что тексель (элемент текстуры) генерируется из памяти текстур для каждого фрагмента и применяется к конкретному фрагменту. После этого могут применяться вычисления тумана, которые сопровождаются тестом ножниц, альфа-тестом, тестом трафарета и тестом буфера глубины (тест буфера глубины представляет собой удаление невидимых поверхностей). Неудачное завершение включенного теста может прекратить длительную обработку квадрата фрагмента. Затем могут быть выполнены операции смешивания цветов, псевдосмешивания (размывания) цветов для передачи полутонов, логической обработки и маскирования с помощью битовой маски. Наконец, полностью обработанный фрагмент выводится в соответствующий буфер, где он окончательно превращается в пиксель и достигает своего конечного местоположения.

2 Библиотеки, относящиеся к OpenGL

2.1 Библиотека OpenGL

Библиотека OpenGL предоставляет мощный, но примитивный набор команд визуализации, и все изображения более высокого уровня должны быть созданы с

использованием именно этих команд. Кроме этого OpenGL-программы должны использовать основные механизмы системы управления окнами. Существует несколько библиотек, которые позволяют упростить решение ваших задач в области программирования. В число этих библиотек входят следующие:
Библиотека утилит OpenGL (GLU -- OpenGL Utility Library) содержит несколько подпрограмм, которые используют OpenGL-команды низкого уровня для выполнения таких задачи, как установка матриц для определенного ориентирования и проецирования просмотра, выполнение тесселяции многоугольников(разбиение произвольного многоугольника на выпуклые многоугольники) и визуализация поверхности. Эта библиотека предоставляется в качестве составной части каждой реализации библиотеки OpenGL. Составные части библиотекиGLU описываются в Справочнике по OpenGL {OpenGL Reference Manual).
Для каждой оконной системы существует библиотека, которая расширяет функциональные возможности данной оконной системы, чтобы обеспечить поддержку визуализации OpenGL. Для вычислительных машин, которые используют оболочку X Window System, в качестве дополнения к библиотеке OpenGL предоставляется Расширение библиотеки OpenGL для оболочки X Window System(GLX -- OpenGL Extension to the X Window System). Подпрограммы GLX используют префикс glX. Для операционных систем Microsoft Windows 95/98/NT интерфейс операционной системы Windows к библиотеке OpenGL обеспечивается подпрограммами библиотеки WGL. Все подпрограммы WGL используют префикс wgl. Для операционной системы OS/2 корпорации IBM используются PGL --интерфейс Администратора представлений (Presentation Manager) к библиотеке OpenGL, и его подпрограммы используют префикс pgl. Для компьютеров фирмы Apple интерфейсом для графических систем, поддерживающих библиотеку OpenGL, является AGL, и подпрограммы AGL используют префикс agl.
Все эти библиотеки расширения оконных систем более подробно описываются в Приложении С. Кроме этого, подпрограммы GLX также описываются в Справочнике по OpenGL.
Инструментарий утилит библиотеки OpenGL (GLUT -- Graphics Library UtilityToolkit) представляет собой независимый от оконной системы инструментарий, написанный Марком Килгардом (Mark Kilgard) для того, чтобы скрыть сложность программного интерфейса прикладных программ (API -- ApplicationProgramming Interface) различных оконных систем. Инструментарий GLUT является предметом следующего раздела, но более подробно он описывается в книге Марка Килгарда OpenGL Programming for the X Window System (ISBN 0-201-48359-9). Подпрограммы GLUT используют префикс glut
FSG (Fahrenheit Scene Graph) представляет собой объектно-ориентированный набор инструментальных средств, основанный на библиотеке OpenGL и предоставляющий объекты и методы для создания интерактивных трехмерных графических прикладных программ. FSG написан на языке программирования С++,содержит предварительно подготовленные объекты и встроенную модель обработки событий при взаимодействии с пользователем, компоненты прикладных программ высокого

уровня для создания и редактирования трехмерных сцен и возможности для обмена данными в других графических форматах. FSG не зависим от OpenGL.

2.2. Подключаемые файлы

Для всех OpenGL-приложений вы можете подключить заголовочный файл gl.h в каждый файл проекта. Почти все OpenGL-приложения используют GLU, вышеупомянутую Библиотеку утилит OpenGL, которая требует включения заголовочного файла glu.h. Так что почти каждый исходный файл OpenGL-приложения начинается со следующих строк:

#include <gi/gl.h> «include <gl/glu.h>

Операционная система Microsoft Windows требует, чтобы заголовочный файл windows.h был включен до подключения заголовочных файлов gl.h или glu.h, поскольку некоторые макрокоманды, определенные в Microsoft Windows-версиях заголовочных файлов gl.h и glu.h, определяются в заголовочном файле windows.h. Если вы обращаетесь непосредственно к библиотеке оконного интерфейса, которая обеспечивает поддержку OpenGL, например, к библиотеке GLX, AGL, PGL или WGL, то необходимо включить дополнительные заголовочные файлы. Например, если вы вызываете библиотеку GLX, то, возможно, потребуется добавить к вашему программному коду строки, приведенные ниже:

«include <X11/Xlib.h> «include <GL/glx.h>

В операционной системе Microsoft Windows для подключения подпрограмм WGL следует добавить к вашему программному коду следующую строку:

«include <windows.h>

(и строку

#include <GL/glaux.h>

если вам нужны расширенные возможности OpenGL. Примечание научн. редактора.)

Если вы используете библиотеку GLUT для управления задачами оконного менеджера, то необходимо будет включить следующую строку:

#include <GL/glut.h>

Большинство OpenGL-приложений также используют системные вызовы для стандартной библиотеки языка программирования С, поэтому обычно следует включать заголовочные файлы, не связанные с обработкой графики (если вы программируете консольное \Ут32-приложение на языке программирования С/С-н--прим. научн. ред.), такие как:

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

2.3 GLUT, инструментарий утилит библиотеки OpenGL

2.3.1. Инструментарий библиотек

Как вы уже знаете, библиотека OpenGL содержит команды визуализации, однако она разрабатывалась таким образом, чтобы быть независимой от любой конкретной оконной или операционной системы. Следовательно, эта библиотека не содержит никаких команд для открытия окон или считывания событий от клавиатуры или мыши. К сожалению, невозможно написать законченную графическую программу, не открывая, по крайней мере, одно окно, а наиболее интересные программы требуют определенного объема обработки данных, вводимых пользователем, или других

услуг от оконной или операционной системы. Во многих случаях законченные программы дают наиболее интересные примеры, поэтому настоящая книга использует библиотеку GLUT для того, чтобы упростить процедуры открытия окон, обнаружения ввода данных пользователем и т.д. Если на вашей системе имеется реализация библиотеки OpenGL и инструментария GLUT, то примеры, приведенные в данной книге, должны работать без изменений при связывании с вашими библиотеками OpenGL и GLUT.

Кроме того, поскольку состав команд рисования в библиотеке OpenGL ограничен только командами, которые генерируют простые геометрические примитивы (точки, линии и многоугольники), библиотека GLUT включает несколько подпрограмм, создающих более сложные трехмерные объекты, такие как сфера, тор и чайник. Таким образом, можно получить для просмотра достаточно интересные кадры вывода программы. (Обратите внимание на то, что Библиотека утилит OpenGL, GLU, также имеет в своем составе подпрограммы построения двумерных поверхностей, которые создают некоторые трехмерные объекты, такие же, как и те, которые создает инструментарий GLUT, в том числе сферу, цилиндр или конус.) Инструментарий GLUT может быть не достаточным для полнофункциональных OpenGL-приложений, но он может оказаться хорошей отправной точкой для изучения OpenGL. Остальная часть настоящего раздела кратко описывает небольшое подмножество подпрограмм библиотеки GLUT таким образом, чтобы вы смогли разобраться с примерами программирования в остальной части данной книги.

### 2.3.2 Управление окнами

Пять подпрограмм инструментария GLUT выполняют задачи, необходимые для того, чтобы инициализировать окно.

Подпрограмма glutlnit(int *argc, char **argv) инициализирует библиотеку GLUT иобрабатывает любые аргументы командной строки (для оболочки X WindowSystem это могли бы быть такие опции, как -display и -geometry). Подпрограмма glutlnit() должна быть вызвана перед любой другой подпрограммой библиотеки GLUT.

Подпрограмма glutInitDisplayMode(unsigned int mode) определяет, какую цветовую модель следует использовать: режим RGBA или режим индексации цвета. Можно также определить, хотите ли вы работать с буфером кадра окна с одинарной или с двойной буферизацией. (Если вы работаете в режиме индексации цвета, то вы, возможно, захотите загрузить некоторые цвета в таблицу компонентов цвета; для того чтобы сделать это, воспользуйтесь командой glutSetColor().) Наконец, можно использовать эту подпрограмму для того, чтобы указать, что вы хотите связать с данным окном буферы глубины, трафарета и/или буфер-накопитель. Например, если вы хотите использовать окно с двойной буферизацией, цветовой моделью RGBA и буфером глубины, то для этого можно вызвать рассматриваемую подпрограмму со следующими параметрами: g\utlnitmsv\&yMote{GLUT_DOUBLE\ GLUT^RGB \ GLUTJ)EPTH).

Подпрограмма glutInitWindowPosition(int x, int y) определяет местоположение левого верхнего угла создаваемого окна на экране монитора.

Подпрограмма glutInitWindowSize(int width, int size) определяет размер создаваемого

окна в пикселях.
Подпрограмма int glutCreateWindow(char *string) создает окно с контекстомOpenGL. Эта подпрограмма возвращает уникальный идентификатор для нового окна. Имейте в виду: до тех пор, пока вызывается подпрограмма glutMainLoopO, это окно еще не отображается на экране.
2.3.3 Функция обратного вызова отображения
Подпрограмма glutDisplayFunc(void (*/«nc)(void)) представляет собой первую и наиболее важную функцию обратного вызова по событию, с которой вам предстоит столкнуться. Всякий раз, когда библиотека GLUT определяет, что содержимое данного окна должно быть восстановлено, выполняется функция обратного вызова, зарегистрированная подпрограммой glutDisplayFunc(). Поэтому вы должны поместить все подпрограммы, которые необходимы для перерисовки сцены, в данную функцию обратного вызова отображения.
Если ваша программа изменяет содержимое окна, то иногда вы должны будете вызывать подпрограмму glutPostRedisplay(void), которая вынуждает подпрограмму glutMainLoopO вызывать зарегистрированную функцию обратного вызова отображения при следующем удобном случае.
2.3.4. Исполнение программы
Самое последнее, что вы должны сделать, это вызвать подпрограмму glutMainLoop(void). При этом отображаются все окна, которые были созданы, и в этих окнах теперь работает визуализация. Начинается обработка событий, и вызывается зарегистрированная функция обратного вызова отображения. Войдя однажды в этот цикл, из него не выходят никогда!
Пример 2 демонстрирует, как можно было бы воспользоваться инструментарием GLUT, чтобы создать простую программу, показанную ранее в примере 1. Обратите внимание на реструктурирование программного кода. Для того чтобы сделать эффективность программы максимальной, все операции, которые должны вызываться однократно (установка цвета фона и системы координат), теперь включены в состав процедуры, названной init(). Операции, необходимые для визуализации (и, возможно, для повторной визуализации) сцены, включены в состав процедуры display(), которая представляет собой зарегистрированную функцию обратного вызова отображения библиотеки GLUT.
Пример 2 Простая программа OpenGL, использующая инструментарий GLUT: hello.c

```
#include <GL/glut.h> #include <stdlib.h>
void display(void)
/* Очистить все пиксели */
glClear(GL_COLOR_BUFFER_BIT);
/* нарисовать белый многоугольник (прямоугольник) с углами,
расположенными в точках с координатами (0.25, 0.25, 0.0)
и (0.75, 0.75, 0.0)*/
glColor3f(1.0, 1.0, 1.0); glBegin(GL_POLYGON);
glVertex3f(0.25, 0.25, 0.0);
glVertex3f(0.75, 0.25, 0.0);
```

```
glVertex3f(0.75, 0.75, 0.0);
glVertex3f(0.25, 0.75, 0.0);glEnd() ;
/* Не ждать! Запустить обработку буферизированных
* подпрограмм OpenGL*/
glFlushO ; }
void init(void) {
/* Выбрать цвет очистки (цвет фона) */
glClearColor (0.0, 0.0, 0.0, 0.0);
/* Инициализировать просматриваемые значения */
glMatrixMode(GL_PROJECTION);
glLoadldentity();
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0); } /*
```

Объявить начальный размер окна, его положение на экране и режим отображения (одинарная буферизация и режим RGBA).

Открыть окно со словом "hello" в строке заголовка. Вызвать подпрограммы инициализации. Зарегистрировать функцию обратного вызова для отображения графики. Войти в основной цикл и обрабатывать события.*/

```
int main(int argc, char** argv) {
glutInit(Sargc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(250, 250);
glutInitWindowPosition(100, 100);
glutCreateWindow("hello");
init();
glutDisplayFunc(display);
glutMainLoop();
return 0; /* Язык программирования С, согласно ISO, требует, чтобы функция main возвращала значение типа int. */ }
```

2.3.4 Обработка событий ввода данных пользователем

Для того чтобы зарегистрировать обратный вызов команд, которые вызываются в том случае, когда происходят указанные события, можно воспользоваться следующими подпрограммами.

Подпрограмма glutReshapeFunc(void (*func)(int w, int h)) указывает на то, какое именно действие должно быть выполнено при изменении размера окна.

Подпрограммы glutKeyboardFunc(void (*/«wc)(unsigned char key, int x, int y)) иglutMouseFunc (void (*func)(int button, int state, int x, int y)) позволяют связывать определенную клавишу клавиатуры или кнопку мыши с подпрограммой, которая вызывается, когда данная клавиша или кнопка мыши нажимается или отпускается пользователем.

Подпрограмма glutMotionFunc(void (*func)(int x, int y)) регистрирует некоторую подпрограмму для обратного вызова при перемещении мыши с нажатой кнопкой.

2.3.5 Управление фоновым процессом

Можно определить некоторую функцию, которая должна быть выполнена с

помощью подпрограммы glutIdleFunc(void (*/«nc)(void)) в том случае, если не ожидаются
никакие другие события, например, когда цикл обработки событий в противном случае перешел бы в состояние простоя. Эта подпрограмма в качестве своего единственного параметра принимает указатель на данную функцию. Для того чтобы отключить выполнение этой функции, передайте ей значение NULL (нуль).

2.3.6 Рисование трехмерных объектов

Библиотека GLUT включает в себя несколько подпрограмм для рисования перечисленных ниже трехмерных объектов:

Конус

Икосаэдр

Чайник

Куб

Октаэдр

Тетраэдр

Додекаэдр

Сфера

Top

Вы можете нарисовать эти объекты в виде каркасных моделей или в виде сплошных закрашенных объектов с определенными нормалями к поверхностям. Например, подпрограммы для куба и сферы имеют следующий синтаксис:
void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
Все эти модели рисуются центрированными относительно начала мировой системы координат.
3. Анимация
3.1. Анимация компьютерной графики
Одна из наиболее захватывающих вещей, которую вы можете сделать в области компьютерной графики, -- это рисование движущихся изображений. Вне зависимости от того, являетесь ли вы инженером, пытающимся увидеть все стороны разрабатываемого механического узла, пилотом, изучающим с использованием моделирования процесс пилотирования самолета, или же просто страстным любителем компьютерных игр, очевидно, что анимация является важной составной частью компьютерной графики.
В кинотеатре иллюзия движения достигается за счет использования последовательности изображений и проецирования их на экран с частотой 24 кадра в секунду. Каждый кадр последовательно перемещается в положение позади объектива, затвор открывается, и данный кадр отображается на экране. Затвор на мгновение закрывается, в то время как пленка протягивается к следующему кадру, затем на экране отображается этот следующий кадр, и так далее. Хотя каждую секунду вы наблюдаете на экране 24 различные кадра, ваш мозг смешивает все эти кадры в "непрерывную" анимацию. (Старые фильмы Чарли Чаплина снимались с частотой 16 кадров в секунду и при воспроизведении фигуры двигались заметными резкими толчками.) Экран в компьютерной графике обычно обновляется (перерисовывает изображение) приблизительно от 60 до 76 раз в секунду, а иногда прикладные программы обеспечивают даже приблизительно 120 обновлений в секунду. Очевидно, что анимация с частотой 60 кадров в секунду выглядит более "гладкой", чем при частоте 30 кадров в секунду, а 120
кадров в секунду заметно лучше, чем 60 кадров в секунду. Однако частоты регенерации, превышающие 120 кадров в секунду, могут быть за пределами точки уменьшения повторного появления, в зависимости от пределов восприятия.
Основная причина того, что технология проецирования кинофильма работает,

заключается в том, что каждый кадр является законченным в момент его отображения на экране. Предположим, что вы пытаетесь сделать компьютерную анимацию из своего кинофильма, состоящего из одного миллиона кадров, с помощью программы, подобной приведенному ниже фрагменту псевдокода:

```
открыть окно();
for (i = 0; i < 1000000; i++) {
очистить окно();
нарисовать_кадр (i) ;
подождать_пока_не_закончится_интервал_в_1_24__долю_секунды(); )
```

Если вы добавите время, которое требуется вашей вычислительной системе для того, чтобы очистить экран и нарисовать типичный кадр, то приведенная выше программа показывает все более тревожащие результаты в зависимости от того, насколько близко подходит время, требуемое ей для очистки экрана и прорисовки кадра к 1/ 24 доле секунды. Предположим, что процедура рисования в этой программе почти полностью занимает 1/24 долю секунды. Элементы, нарисованные в самом начале, видимы в течение полной 1/24 доли секунды и представляют сплошное изображение на экране; элементы, нарисованные в конце рассматриваемого интервала, немедленно очищаются, как только программа запускается для рисования следующего кадра. Они представляют собой в лучшем случае некое подобие призрачного изображения, поскольку большую часть интервала в 1/24 секунды ваш глаз рассматривает очищенный фон вместо тех элементов, которые, к несчастью для них, были нарисованы последними. Проблема в данном случае заключается в том, что приведенная выше программа не отображает полностью нарисованные кадры; вместо этого вы наблюдаете процесс рисования в его развитии.

Большинство реализаций библиотеки OpenGL обеспечивает двойную буферизацию -- аппаратную или программную, которая предоставляет два готовых буфера с цветными изображениями. Изображение из одного буфера отображается на экране, в то время как в другом буфере рисуется новое изображение. Когда рисование очередного кадра завершается, эти два буфера меняются местами, и тот буфер, что содержал отображаемое изображение, теперь используется для рисования, и наоборот. Это похоже на работу кинопроектора, пленка в котором содержит всего два кадра и склеена в петлю; в то время как один проецируется на экран, киномеханик отчаянно стирает и перерисовывает невидимый зрителю кадр. Если киномеханик работает достаточно быстро, то зритель не замечает различий между таким "кинопроектором" и реальной системой, в которой все кадры уже нарисованы, и кинопроектор просто отображает их один за другим. При использовании двойной буферизации каждый кадр отображается только тогда, когда его рисование завершено; зритель никогда не увидит частично нарисованного кадра.

Псевдокод измененной версии приведенной выше программы, которая отображает плавно анимированную графику, используя при этом двойную буферизацию, мог бы выглядеть следующим образом:

```
открыть_окно_в_режиме_двойной_буфериэации(); for (i = 0; i < 1000000; i++) {
```

очис1ить_окно();
нарисовать_кадр(i);
поменять_буферы_местами() ; }
3.2 Обновление отображаемой информации во время паузы
Для некоторых реализаций библиотеки OpenGL в дополнение к простой перемене мест отображаемого и рисуемого буферов, подпрограмма поменять_буферы_местами() ожидает, пока не закончится текущий период обновления отображаемой информации для того, чтобы информация из предыдущего буфера был отображена полностью. Эта подпрограмма также позволяет новому буферу быть полностью отображенным, начиная с начала.
Принимая, что ваша вычислительная система обеспечивает обновление отображения 60 раз в секунду, получим, что максимальная скорость передачи кадров, которой можно достичь -- 60 кадров в секунду (fps -- frames per second). Это означает, что, если все кадры могут быть очищены и нарисованы в течение 1/60 доли секунды, то ваша анимация при данной скорости будет воспроизводиться плавно.
В подобных системах часто бывает так, что кадр оказывается слишком сложным для того, чтобы быть нарисованным в течение 1/60 доли секунды, и, таким образом, каждый кадр отображается более одного раза. Если, например, требуется 1/45 доля секунды для того, чтобы нарисовать некоторый кадр, вы задаете скорость передачи кадров равной 30 fps, и тогда графическое изображение "простаивает" в течение 1/30 -- 1/45 = 1/90 доли секунды на каждый кадр, или треть всего времени отображения.
Кроме того, частота обновления отображаемой видеоинформации является постоянной величиной, которая может иметь некоторые неожиданные последствия с точки зрения производительности. Например, при периоде обновления информации, отображаемой на мониторе, равной 1/60 доли секунды и при постоянной скорости передачи кадров вы можете работать со скоростями 60 fps, 30 fps, 20 fps, 15 fps, 12 fps и т. д. (60/1, 60/2, 60/3, 60/4, 60/5, и т. д.). Это означает, что если вы пишете прикладную программу и постепенно добавляете к ней новые функциональные возможности (предположим, что эта программа -- имитатор полета, и вы добавляете наземный пейзаж), то сначала каждая новая добавляемая деталь не будет оказывать никакого эффекта на суммарную производительность -- вы все равно получаете скорость передачи кадров, равную 60 fps. Затем, когда вы добавляете еще одну новую деталь, система уже не может нарисовать все это в течение 1/60 доли секунды, и анимация резко замедляется -- с 60 fps до 30 fps, поскольку она пропускает первый возможный момент смены буферов. Аналогичная ситуация происходит, когда время рисования одного кадра становится больше, чем 1/30 доля секунды -- скорость передачи кадров анимации скачком уменьшается от 30 fps до 20 fps.
Если сложность сцены такова, что время ее рисования оказывается близко к любому из этих "волшебных" интервалов времени (1/60 доля секунды, 2/60 доли секунды, 3/60 доли секунды и т. д. в рассматриваемом примере), то из-за наличия случайного изменения некоторые кадры будут идти немного быстрее, а некоторые -- немного

медленнее. В результате скорость передачи кадров становится непостоянной, что визуально может быть неприятно. В этом случае, если вы не можете упростить сцену так,
чтобы все кадры рисовались достаточно быстро, возможно, лучше было бы преднамеренно добавить крошечную задержку для того, чтобы иметь уверенность, что все кадры пропущены, задавая при этом постоянную, более медленную скорость передачи кадров. Если же кадры имеют существенно отличающуюся степень сложности, тогда, возможно, вам потребуется более сложный подход к решению этой проблемы.
Движение = Перерисовка изображения + Перестановка буферов
Структура реальных программ анимации не слишком отличается от приведенного описания. Обычно проще перерисовать буфер целиком с чистого листа для каждого кадра, чем выяснять, какие части кадра требуют изменения. Это положение особенно справедливо для таких прикладных программ, как трехмерные имитаторы полета, где самое малое изменение ориентации самолета изменяет позицию всего вида из окна пилотской кабины.
В большинстве видов анимации объекты на сцене просто перерисовываются с различными преобразованиями: перемещается точка наблюдения зрителя или автомобиль немного проезжает по дороге, или какой-нибудь объект поворачивается на небольшой угол. Если для операций, не связанных с рисованием, требуется существенный объем повторных вычислений, то достижимая скорость передачи кадров часто замедляется. Следует иметь в виду, однако, что для выполнения таких вычислений после выполнения подпрограммы поменять_буферы_местами() часто может использоваться время простоя.
Библиотека OpenGL не имеет в своем составе команды типа "поменять_буферы_-местами()", поскольку такая функциональная возможность могла бы быть доступна не для всех аппаратных средств и, в любом случае, ее реализация существенно зависит от используемой оконной системы. Например, если вы используете оболочку X Window System и обращаетесь к ней непосредственно, то можно было бы воспользоваться следующей подпрограммой из библиотеки GLX:
void glXSwapBuffers(Display *dpy, Window window);
(Эквивалентные подпрограммы для других оконных систем приведены в Приложении С.)
Если вы используете библиотеку GLUT, то вы, возможно, захотите вызвать следующую подпрограмму:
void glutSwapBuffers(void);
Пример 3 иллюстрирует использование подпрограммы glutSwapBuffers() для рисования вращающегося квадрата, как это показано на рисунке 3. Этот пример также демонстрирует то, как следует использовать библиотеку GLUT для контроля состояния устройства ввода данных, а также для включения и выключения функции простоя. В приведенном примере кнопки мыши включают и отключают вращение квадрата.
Пример 3 Программа с использованием двойной буферизации: double.c

```
#include <stdlib.h> #include <GL/glut.h>
static GLfloat spin = 0.0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel(GL_FLAT); } void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glPushMatrix();
glRotatef(spin, 0.0, 0.0, 1.0);
glColor3f(1.0, 1.0, 1.0);
glRectf(-25.0, -25.0, 25.0, 25.0);
glPopMatrix () ;
glutSwapBuffers () ; )
void spinDisplay(void)
{
spin = spin +2.0;
if (spin > 360.0)
spin = spin - 360.0;
glutPostRedisplay(); }
void reshape (int w, int h)
{
glViewport(0, 0,  (GLsizei) w,  (GLsizei) h) ;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
void mouse(int button, int state, int x, int y) {
switch (button) {
case GLUT_LEFT_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(spinDisplay); break; case GLUT_MIDDLE_BUTTON:
if (state == GLUT_DOWN)
glutIdleFunc(NULL); break; default:
break; } }
/*
Запросить режим отображения с двойной буферизацией.
Зарегистрировать функции обратного вызова по вводу данных от мыши*/
int main(int argc, char** argv) {
glutInit(Sargc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize(250, 250);
```

```
glutInitWindowPosition(100,  100);
glutCreateWindow(argv[0]);
init<);
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
return 0 ; }
```

Пример программы, написанной на C, с применением OpenGL.
Программа визуализирует вращающийся куб. На куб наложены текстуры, возможно включение альфа-прозрачности во время визуализации.

```
#include <GL/glut.h>
#include <GL/glaux.h>
#include <GL/glu.h>
int tex[1];
float xtr,ytr,ztr;
float rotx,roty,rotz;
float LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f };
float LightPosition[]= { 0.0f, 0.0f, 5.0f, 1.0f };//позиция освещения
float LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f }; //рассеивание
void LoadTexs()//инициализация и загрузка текстур
{
 AUX_RGBImageRec *tex1;
 tex1 = auxDIBImageLoad("droy.bmp");
 glGenTextures(1, &tex[0]);
 glBindTexture(GL_TEXTURE_2D, tex[0]);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, 3, tex1->sizeX, tex1->sizeY, 0,
 GL_RGB, GL_UNSIGNED_BYTE, tex1->data);
}
void init()
{
 LoadTexs();//процедура загрузки текстуры
 glColor4f(1.0f,1.0f,1.0f,0.5f);
 glBlendFunc(GL_SRC_ALPHA,GL_ONE); // параметры смешивания
 glEnable(GL_TEXTURE_2D);
 glClearColor (0.0, 0.0, 0.0, 0.0);// цвет фона
 glClearDepth(1.0);
 glDepthFunc(GL_LESS);
 glEnable(GL_DEPTH_TEST); //тест глубины
 glShadeModel (GL_SMOOTH);
 glTranslatef(0.0f,0.0f,-5.0f);
```

```
 glLightfv(GL_LIGHT1,GL_AMBIENT,LightAmbient); //параметры освещения
 glLightfv(GL_LIGHT1,GL_POSITION,LightPosition);
 glLightfv(GL_LIGHT1,GL_DIFFUSE,LightDiffuse);
 glEnable(GL_LIGHT1);//освещения
 glEnable(GL_LIGHTING);
 glEnable (GL_COLOR_MATERIAL);
 glColorMaterial (GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);//очистка буфера глубины
 glPushMatrix();//сохранение матрицы
 glTranslatef(xtr,ytr,ztr);
 glRotatef(rotx,1.0,0.0,0.0);
 glRotatef(roty,0.0,1.0,0.0);
 glRotatef(rotz,0.0,0.0,1.0);
glBegin(GL_QUADS);
glNormal3f( 0.0f, 0.0f, 1.0f);// Передняя грань
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Низ лево
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Низ право
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Верх право
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Верх лево
```