Лекция 19
Введение в OpenGL

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.

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.

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.

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.

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.

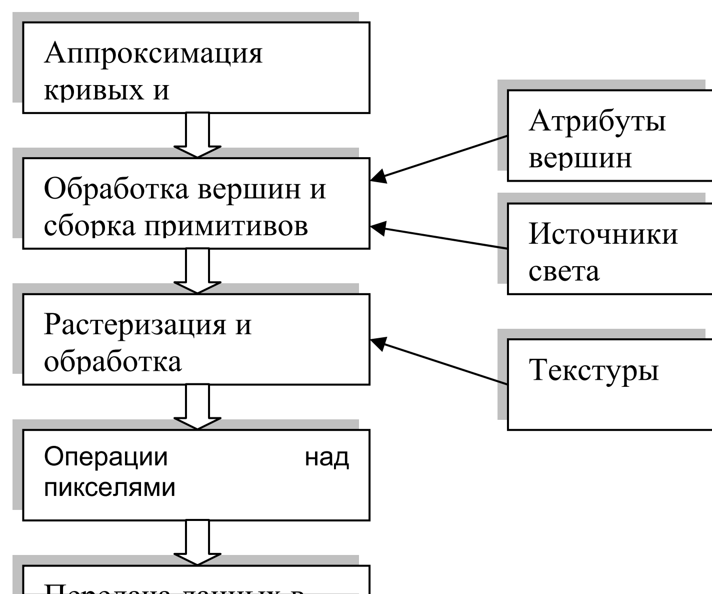


Рис. 1. Функционирование конвейера OpenGL

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.

Синтаксис команд

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

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

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 C.

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.

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.

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. 0). 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.

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.

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.

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.

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.

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();
```

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.**

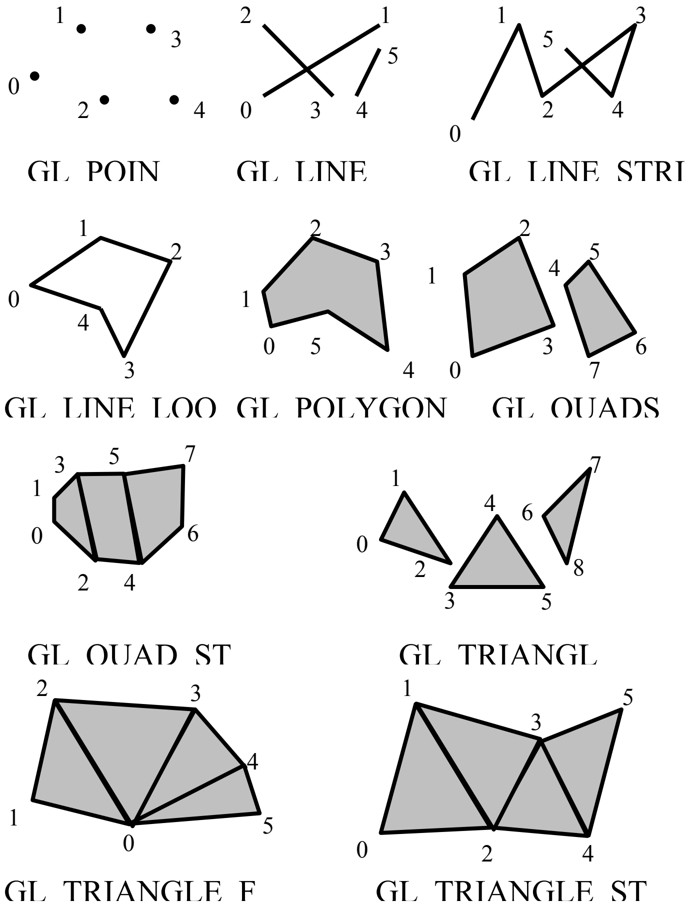


Рис. 2. Примитивы OpenGL

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.

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.

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)
```

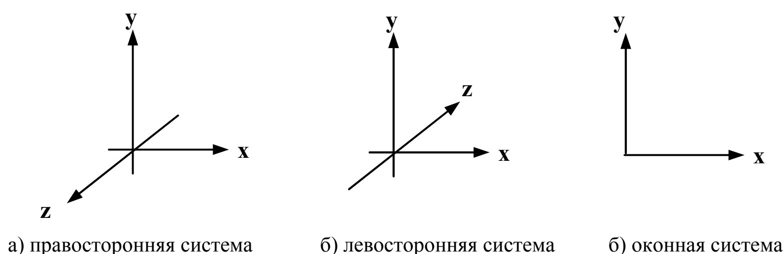
где *indices* – это массив номеров вершин, которые надо использовать для построения примитивов, *type* определяет тип элементов этого массива: **GL_UNSIGNED_BYTE**, **GL_UNSIGNED_SHORT**, **GL_UNSIGNED_INT**, а *count* задает их количество.

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.

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 0. 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. 3 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.

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.

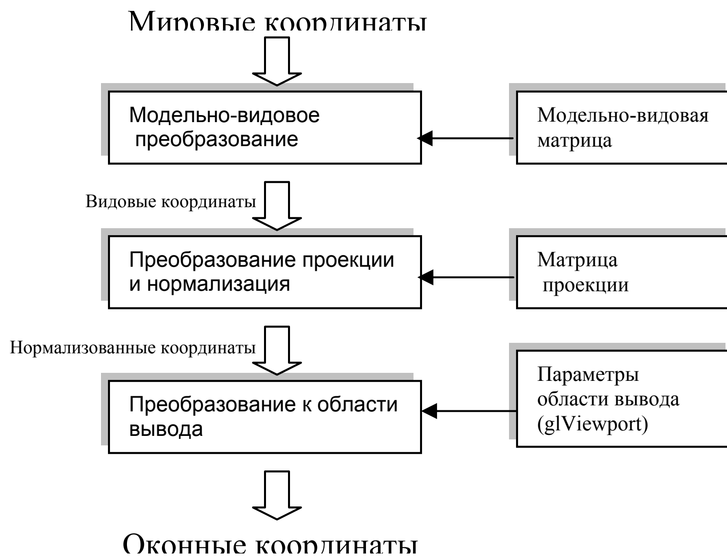


Рис. 4. Преобразования координат в OpenGL

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.

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 у, 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.

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)
```

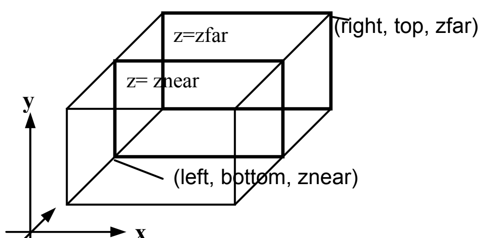


Рис. 5 Ортографическая проекция

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.

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) 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. 0.).

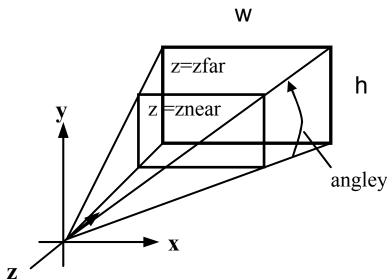


Рис. 6 Перспективная проекция

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

Например:

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

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.

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 0 режим нормализации 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).

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. **Ошибка! Источник ссылки не найден.**.

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);
```

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). 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.

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.

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. 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.

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)

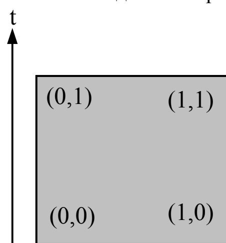


Рис. 7 Текстурные координаты

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.

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**.

Пример:

Рассмотрим, как можно задать зеркальную текстуру. При таком наложении текстуры изображение будет как бы отражаться от поверхности объекта, вызывая интересный оптический эффект. Для этого сначала надо создать два целочисленных массива коэффициентов `s_coeffs` и `t_coeffs` со значениями (1,0,0,1) и (0,1,0,1) соответственно, а затем вызвать команды:

```
glEnable (GL_TEXTURE_GEN_S);
glTexGeni (GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGendv (GL_S, GL_EYE_PLANE, s_coeffs);
```

и такие же команды для координаты *t* с соответствующими изменениями.

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.

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*.

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.