

# OpenGL

## Компьютерная графика

Издательство ТГТУ

УДК 004.92(075)  
ББК Ж11я73  
В191

Рецензенты:

Доктор технических наук, профессор ТГТУ  
*А.А. Чуриков*

Коммерческий директор ООО "Бит"  
*А.В. Крюков*

**Васильев, С.А.**

В191 OpenGL. Компьютерная графика : учебное пособие / С.А. Васильев. Тамбов : Изд-во Тамб. гос. техн. ун-та, 2005. 80 с.

Данное учебное пособие призвано послужить как в качестве учебника по открытой графической библиотеке, так и в качестве справочника по использованию основных команд OpenGL. Оперирова предметным указателем можно быстро найти то или иной материал по использованию графических функций, процедур и символьных констант.

Пособие призвано помочь студентам в выполнении курсовых работ и проектов, а также в их научно-исследовательской работе, где требуется организовать высокоэффективную визуализацию 2D

или 3D графики.

Предназначено для студентов 3, 4 курсов дневной формы обучения по специальности 230104.

УДК 004.92(075)

ББК Ж11я73

**ISBN 5-8265-0417-X**

© Васильев С.А., 2005

© Тамбовский государствен-  
ный

технический университет  
(ТГТУ), 2005

Министерство образования и науки Российской Федерации

Государственное образовательное учреждение  
высшего профессионального образования

**"Тамбовский государственный технический университет"**

OpenGL

Компьютерная графика

Утверждено Ученым советом ТГТУ  
в качестве учебного пособия



---

Тамбов  
Издательство ТГТУ  
2005

Учебное издание

**ВАСИЛЬЕВ Сергей Александрович**

OpenGL  
Компьютерная графика

Учебное пособие

Редактор Т.М. Федченко

Компьютерное макетирование И.В. Евсеевой

Подписано к печати 03.06.2005

Гарнитура Times New Roman. Формат 60 × 84/16. Бумага офсетная.

Печать офсетная. Объем: 4,65 усл. печ. л.; 4,62 уч.-изд. л.

Тираж 200 экз. С. 421

Издательско-полиграфический центр ТГТУ  
392000, Тамбов, Советская, 106, к. 14

## ВВЕДЕНИЕ

Стремительный рост интереса к компьютерной графике во многих сферах деятельности человека способствует развитию ее математических и алгоритмических основ. Соответственно видоизменяется и стандартизация в области компьютерной графики. Наблюдается тенденция смены общепринятых стандартов CORE, GKS, GKS-3D, PHIGS на стандарт открытой графической библиотеки – OpenGL (Open Graphic Library) – являющимся базовым стандартом для большинства рабочих графических станций в мире (Sun, Silicon Graphics и т.п.).

Стандарт OpenGL был разработан и утвержден в 1992 году ведущими фирмами в области разработки программного обеспечения, среди которых Digital Equipment Corporation, Evans & Sutherland, Hewlett Packard Co., IBM Corp., Intel Cor., Intergraph Cor., Silicon Graphics Inc., Sun Microsystems и Microsoft. Основой его стала библиотека IRIS GL, разработанная Silicon Graphics.

Характерные особенности OpenGL, обеспечивающие распространение и развитие этого графического стандарта.

• **СТАБИЛЬНОСТЬ.** ВСЕ ВНОСИМЫЕ В НЕГО ДОПОЛНЕНИЯ И ИЗМЕНЕНИЯ РЕАЛИЗУЮТСЯ ПРИ СОХРАННОСТИ СОВМЕСТИМОСТИ С РАЗРАБОТАННЫМ РАНЕЕ ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ.

• **НАДЕЖНОСТЬ.** ПРИЛОЖЕНИЯ, ИСПОЛЬЗУЮЩИЕ OPENGL, ГАРАНТИРУЮТ ОДИНАКОВЫЙ ВИЗУАЛЬНЫЙ РЕЗУЛЬТАТ ВНЕ ЗАВИСИМОСТИ ОТ ТИПА ИСПОЛЬЗУЕМОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ И ОБОРУДОВАНИЯ.

• **ПЕРЕНОСИМОСТЬ.** ПРИЛОЖЕНИЯ МОГУТ ВЫПОЛНЯТЬСЯ КАК НА ПЕРСОНАЛЬНЫХ КОМПЬЮТЕРАХ, ТАК И НА РАБОЧИХ СТАНЦИЯХ И СУПЕРКОМПЬЮТЕРАХ.

• **ПРОСТОТА ИСПОЛЬЗОВАНИЯ.** 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.

OpenGL – не язык программирования, а программный интерфейс приложений. Всякий раз, когда мы говорим, что программное приложение выполнено на OpenGL, то подразумеваем, что оно было написано на некотором языке программирования (например, C++) и делает запросы к одной или более библиотекам OpenGL.

Основные возможности OpenGL, предоставленные разработчикам:

- геометрические примитивы (точки, линии и многоугольники);
- растровые примитивы (битовые массивы пикселей);
- работа с цветом в RGBA и индексном режимах;
- видовые, модельные и текстурные преобразования;
- удаление невидимых линий и поверхностей (z-буфер);
- работа с прозрачностью поверхности многоугольников;
- использования В-сплайнов;
- работа с текстурами;
- применение освещения;
- использование смешивания цветов, устранение ступенчатости (anti-aliasing), моделирование "тумана" и других "атмосферных" эффектов.

Данное учебное пособие предполагается использовать как учебник по открытой библиотеке, и как справочник по использованию основных команд OpenGL. Поможет предметный указатель быстро найти тот или иной материал по использованию графических функций, процедур и символьных констант. Описание многих команд OpenGL сопровождается фрагментами программных кодов на языке C++, показывающих их практическое использование.

Кроме этого, данное пособие призвано помочь студентам в выполнении курсовых работ и проектов, а также в их научно-исследовательской работе, где требуется организовать высокоэффективную визуализацию 2D или 3D графики.

## 1 ИНТЕРФЕЙС OPENGL

Интерфейс OpenGL реализован в виде набора команд (функций и процедур), которые позволяют программисту организовывать высококачественную визуализацию 2D или 3D графику.

Все базовые функции OpenGL хранятся в основной библиотеке GL и нескольких дополнительных, например, библиотеке утилит – GLU (GL Utility).

Функции библиотеки GLU определены через базовые функции GL. Благодаря GLU можно легко строить такие геометрических примитивы, как куб, шар, цилиндр, диск. Кроме этого, GLU предоставляет функции построения сплайнов, реализацию дополнительных операций над матрицами и т.п.

OpenGL не включает в себя никакие специальные команды для работы с окнами или вводом информации от пользователя. Программист должен самостоятельно организовывать связь своих графических приложений с оконными подсистемами и обеспечивать взаимодействия с функциями пользователя. Для облегчения подобной работы программиста были созданы специальные 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). Официально GLUT не входит в OpenGL, но ее легко найти в Интернете. Более того, имеются свои реализации GLUT для различных операционных систем.

### 2 СТРУКТУРА ГРАФИЧЕСКОЙ ПРОГРАММЫ OPENGL ДЛЯ СРЕДЫ WINDOWS

**ДЛЯ РАБОТЫ OPENGL В ОПЕРАЦИОННОЙ СИСТЕМЕ WINDOWS ИСПОЛЬЗУЕТСЯ ПОНЯТИЕ КОНТЕКСТА ВОСПРОИЗВЕДЕНИЯ (RENDERING CONTEXT), КОТОРЫЙ СВЯЗЫВАЕТ OPENGL С ОКОННОЙ СИСТЕМОЙ WINDOWS.**

**ПРИЛОЖЕНИЕ, РАБОТАЮЩЕЕ С КОМАНДАМИ OPENGL, ДОЛЖНО СОЗДАТЬ ОДНО ИЛИ НЕСКОЛЬКО КОНТЕКСТОВ ВОСПРОИЗВЕДЕНИЯ ДЛЯ ПОТОКА И СДЕЛАТЬ ОДНО ИЗ НИХ ТЕКУЩИМ. ПЕРЕД СОЗДАНИЕМ КОНТЕКСТА ВОСПРОИЗВЕДЕНИЯ НЕОБХОДИМО УСТАНОВИТЬ ДЛЯ НЕГО ФОРМАТ ПИКСЕЛЕЙ, КОТОРЫЙ ОПРЕДЕЛЯЕТ НЕКОТОРЫЕ СВОЙСТВА ПОВЕРХНОСТИ РИСОВАНИЯ OPENGL. ТАКИМ ОБРАЗОМ, ПРОГРАММЫ, ИСПОЛЬЗУЮЩИЕ OPENGL, СТРОЯТСЯ ПО СЛЕДУЮЩЕЙ СХЕМЕ.**

**1 СОЗДАЕМ ОКНО ПРОГРАММЫ. ЗДЕСЬ НЕОБХОДИМО ОБЯЗАТЕЛЬНО УСТАНОВИТЬ СТИЛЬ ОКНА ФУНКЦИЕЙ CREATEWINDOW ЧЕРЕЗ ПАРАМЕТРЫ WS\_CLIPCHILDREN И WS\_CLIPSIBLINGS.**

**2 ПОСЛЕ УСПЕШНОГО СОЗДАНИЯ ОКНА НЕОБХОДИМО ОТКРЫТЬ КОНТЕКСТ ОТОБРАЖЕНИЯ (RENDERING CONTEXT). РЕКОМЕНДУЕТСЯ ОТКРЫВАТЬ КОНТЕКСТ ОТОБРАЖЕНИЯ ВО ВРЕМЯ ОБРАБОТКИ СООБЩЕНИЯ WM\_CREATE.**

**3 ДЛЯ СОЗДАНИЯ КОНТЕКСТА ОТОБРАЖЕНИЯ НЕОБХОДИМО ПРЕДВАРИТЕЛЬНО ОТКРЫТЬ КОНТЕКСТ ОКНА (HDC), НАПРИМЕР, ФУНКЦИЕЙ GETDC.**

**4 ПРИ НАСТРОЙКЕ ХАРАКТЕРИСТИК КОНТЕКСТА ОТОБРАЖЕНИЯ УСТАНОВИТЬ СООТВЕТСТВУЮЩИЕ ЗНАЧЕНИЯ ПОЛЕЙ СТРУКТУРЫ PIXELFORMATDESCRIPTOR И ВЫЗВАТЬ ФУНКЦИЮ CHOOSEPIXELFORMAT. ЭТА ФУНКЦИЯ ВОЗВРАЩАЕТ НОМЕР ПИКсельного ФОРМАТА, КОТОРЫЙ МОЖНО ИСПОЛЬЗОВАТЬ.**

**5 ВЫЗОВОМ ФУНКЦИИ SETPIXELFORMAT ЗАДАТЬ СООТВЕТСТВУЮЩИЙ ПИКСельный ФОРМАТ В КОНТЕКСТЕ HDC.**

**6 НА ОСНОВАНИИ КОНТЕКСТА HDC СОЗДАТЬ КОНТЕКСТ ОТОБРАЖЕНИЯ (HRC) ВЫЗОВОМ ФУНКЦИИ WGLCREATECONTEXT. ДЛЯ ПЕРЕАДРЕСАЦИИ ТЕКУЩЕГО ВЫВОДА ГРАФИКИ OPENGL В HRC НЕОБХОДИМО ВЫЗЫВАТЬ ФУНКЦИЮ WGLMAKECURRENT.**

**7 В ТЕКУЩИЙ КОНТЕКСТ ОТОБРАЖЕНИЯ ВЫВЕСТИ ГРАФИЧЕСКИЕ ОБЪЕКТЫ, ИСПОЛЬЗУЯ ФУНКЦИИ ДЛЯ РАБОТЫ С ГРАФИЧЕСКИМИ ПРИМИТИВАМИ OPENGL. ГРАФИЧЕСКИЙ ВЫВОД МОЖНО ОСУЩЕСТВЛЯТЬ ВО ВРЕМЯ ОБРАБОТКИ СООБЩЕНИЯ WM\_PAINT ИЛИ ДРУГИХ НЕОБХОДИМЫХ СООБЩЕНИЙ WINDOWS.**

**8 ПЕРЕД ЗАКРЫТИЕМ ОКНА ПРОГРАММЫ НЕОБХОДИМО ЗАКРЫТЬ ВСЕ ОТКРЫТЫЕ КОНТЕКСТЫ ОТОБРАЖЕНИЯ. ТАКЖЕ СЛЕДУЕТ ЗАКРЫТЬ ВСЕ КОНТЕКСТЫ ГРАФИЧЕСКОГО УСТРОЙСТВА. ЭТО МОЖНО СДЕЛАТЬ В ХОДЕ ОБРАБОТКИ СООБЩЕНИЙ WM\_DESTROY ВЫЗОВОМ ФУНКЦИЙ RELEASEDC И WGLDELETECONTEXT.**

### 3 СИНТАКСИС КОМАНД

Синтаксис полного имени команды имеет вид:

`rt glCom_name[1 2 3 4][b s i f d ub us ui][v]` (type1 *arg1*, ..., typeN *argN*)

Таким образом, имя состоит из нескольких частей:

<b>Gl</b>	Это имя библиотеки, в которой описана эта функция: для базовых функций OpenGL, функций из библиотек GLU, GLUT, GLAUX это <code>gl</code> , <code>glu</code> , <code>glut</code> , <code>aux</code> соответственно.
<b>rt</b>	Определяет тип возвращаемого значения и для каждой команды указывается в явном виде.
<b>Com_name</b>	Имя команды, например <code>glColor</code> или <code>glVertex</code> .
<b>[1 2 3 4]</b>	Цифра, показывающая число аргументов команды.
<b>[b s i f d ub us ui]</b>	Символы, определяющие тип аргумента: символ <b>b</b> означает тип <code>GLbyte</code> (аналог <code>char</code> в C/C++), символ <b>f</b> – тип <code>GLfloat</code> (аналог <code>float</code> ), символ <b>i</b> – тип <code>GLint</code> (аналог <code>int</code> ) и так далее. Полный список типов и их описание можно посмотреть в файле <code>gl.h</code> .
<b>[v]</b>	Наличие этого символа показывает, что в качестве параметров функции используется указатель на массив значений.

Компоненты, представленные в квадратных скобках в некоторых названиях не используются.

Например, команда для определения вершины, имеющая вид: `void glVertex3f(float arg1, float arg2, float arg3)` в представленном синтаксисе записывается в виде `void glVertex3[f](atype arg)`.

Благодаря тому, что OpenGL реализован по модели клиент сервер, то имеется возможность управлять процессом рисования примитивов. Для этого в OpenGL существуют две команды:

```
void glFinish()
```

```
void glFlush()
```

Команда `glFinish()` приостанавливает выполнение программы, пока не будут завершены все вызванные перед ней команды OpenGL.

Команда `glFlush()` вызывает немедленное рисование ранее переданных команд.

### 4 ПРОЦЕСС РЕГЕНЕРАЦИИ ИЗОБРАЖЕНИЯ

Основной задачей программы, использующей OpenGL, является обработка трехмерной сцены и интерактивное отображение в буфере кадра ее проекции. Для динамической графики приложение OpenGL в бесконечном цикле должно вызывать функцию регенерации (обновления) изображения.

Обычно функция регенерации изображения состоит из следующих шагов:

- 1 Очистка буферов ( цвета, глубины сцены и т.п.) OpenGL.
- 2 Установка положения наблюдателя (виртуальной камеры).
- 3 Преобразование и рисование геометрических объектов.

Очистка буферов производится с помощью команд:

```
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 интервалу значений [0,1]. Четвертый параметр задает так называемую альфа компоненту (прозрачность). Если этот параметр равен 1, то цвет не прозрачный. По умолчанию цвет – черный (0,0,0,1).

Команда `glClear` очищает буферы, а параметр *buf* определяет какие именно буферы нужно очистить. Данный параметр формируется маску для очистки комбинацией (логической функцией OR) кон-

стант.

Маска	Буфер для очистки
<b>GL_COLOR_BUFFER_BIT</b>	Буфер изображения
<b>GL_DEPTH_BUFFER_BIT</b>	Z-буфер
<b>GL_ACCUM_BUFFER_BIT</b>	Аккумулирующий бу-
<b>GL_STENCIL_BUFFER_BIT</b>	ер Буфер трафарета

Например, в программах для очистки буферов цвета и глубины (Z-буфера) вызывают команду:

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

Установка положения наблюдателя (виртуальной камеры) и преобразования трехмерных объектов (масштабирование, поворот, и т.д.) выполняются с помощью задания соответствующих матриц преобразования. Подробное рассмотрение преобразований объектов и настройки положения виртуальной камеры будут приведены далее.

Для улучшения качества визуализации динамически изменяющихся сцен рекомендуется использовать двойную буферизацию, т.е. необходимо обеспечить перезапись внеэкранного буфера в основной. Для этого предусмотрена команда **BOOL SwapBuffers (HDC hDC)**;

Следует помнить, что после выполнения команды **SwapBuffers(hDC)** содержание внеэкранного буфера не определено.

## 5 ВЫВОД ГЕОМЕТРИЧЕСКИХ ПРИМИТИВОВ

Геометрические объекты в OpenGL задаются вершинами. Вершина – это точка в пространстве графической сцены. Для ее определения в библиотеке OpenGL реализована специальная команда:

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

Вызов любой команды `glVertex*` всегда определяется четырьмя однородными координатами:  $x$ ,  $y$ ,  $z$  и  $w$ . Если вызывается команда `glVertex3*`, то вершина задается  $x$ ,  $y$  и  $z$  координатами, при этом  $w$  полагается равной 1. Для двумерного случая  $z = 0$ , а  $w = 1$ .

Вершины в OpenGL объединяются в графические примитивы. Это может быть фигура, такая как точка, линия, многоугольник, прямоугольник пикселей или битовый массив. Каждая вершина примитива имеет ассоциированные с ней данные.

- *Текущий цвет* – который вместе с условиями освещения определяет результирующий цвет вершины. Цвет задается, например, командой `glColor*` для режима RGBA.
- *Текущая позиция раstra* – используется для определения координат раstra при работе с пикселями и битовыми массивами. Задается командой `glRasterPos*`.
- *Текущая нормаль* – определяет вектор нормали, ассоциированный с отдельной вершиной, и задает ориентацию содержащей ее поверхности в трехмерном пространстве. Для указания нормали используется команда `glNormal*`.
- *Текущие координаты текстуры* – определяют местоположение вершины в карте текстуры. Задается командой `glTexCoord*`.

### 5.1 Задание цветовых атрибутов геометрическим объектам

В OpenGL предусмотрено два режима установки цвета:

- задание индекса в палитру цветов;
- указание непосредственных значений базовых составляющих R (красный), G (зеленый), B (синий) цвета.

В настоящее время графические приложения в основном используют покомпонентное (RGBA) задание цвета. Благодаря этому максимально используются возможности High Color и True Color режимов графических карт. В данных режимах в полной мере моделируется реалистическая графика (работа с

освещением, туманом, прозрачностью, устранение ступенчатости и т.д.).

Для установки цвета в режиме RGBA в OpenGL предусмотрены команды:

`void glColor[3 4][b s I f d](GLtype components)`

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

Первые три параметра команды glColor4\* задают значения R, G, B составляющих цвета. Четвертый параметр (A) определяет "прозрачность" цвета. Независимо от типа задаваемых параметров, все они хранятся

в формате с плавающей точкой и принимают значения из диапазона

[0.0, 1.0]. Значение 1.0 соответствует максимальной интенсивности соответствующего компонента. Для параметра альфа (A) 1.0 соответствует непрозрачному состоянию, а 0.0 – полному прозрачному состоянию цвета. При целочисленных значениях аргументов происходит внутреннее преобразование к формату с плавающей точкой в диапазон [0.0, 1.0] по следующей схеме.

GLtype	Преобразование
GLbyte	$(2c + 1) / (2^8 - 1)$
GLshort	$(2c + 1) / (2^{16} - 1)$
GLint	$(2c + 1) / (2^{32} - 1)$
GLfloat	$c$
GLdouble	$c$

Примечание.  $c$  – значение аргументов.

После того как установлен цвет, его значение распространяется на все последующие графические примитивы.

Если требуется плавный переход цветов от одной вершины примитива к другой, то для этого в OpenGL предусмотрена команда

glEnable(GL\_SMOOTH). В этом случае рассчитывается плавный переход цвета между соседними вершинами по закону линейной интерполяции

$c = t c_1 + (1 - t) c_2$ , где  $c_1$  и  $c_2$  – значения цвета соседних вершин, а  $t$  – коэффициент интерполяции, изменяющийся в диапазоне [0,1].

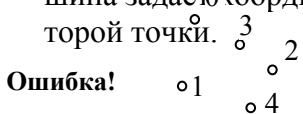
## 5.2 Типовые графические примитивы

Примитивы или группа примитивов определяются внутри командных скобок glBegin/glEnd:

`void glBegin (GLenum mode)`

`void glEnd ()`

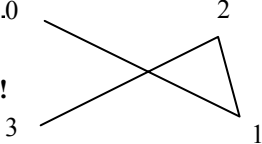
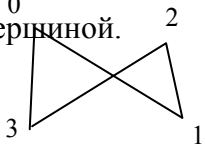
Параметр mode определяет тип примитива, который задается внутри и может принимать следующие значения.

Значение mode	Описание
<b>GL_POINTS</b>	<p>Определяет точки. Каждая вершина задает координаты некоторой точки.</p>  <p>Ошибка!</p>

Продолжение табл.

Значение mode	Описание
---------------	----------

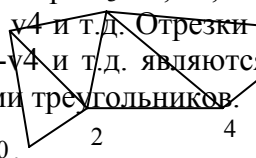
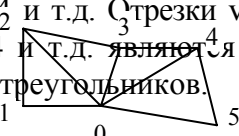


Значение <i>mode</i>	Описание
<b>GL_LINES</b>	<p><b>ФОРМИРУЕТ ОТРЕЗКИ (ОТРЕЗОК) ПРЯМЫХ. КАЖДАЯ ПОСЛЕДОВАТЕЛЬНАЯ ПАРА ВЕРШИН ОПРЕДЕЛЯЕТ ОТРЕЗОК; ЕСЛИ ЗАДАНО НЕЧЕТНОЕ ЧИСЛО ВЕРШИН, ТО ПОСЛЕДНЯЯ ВЕРШИНА ИГНОРИРУЕТСЯ. ПЕРВЫЕ ДВЕ ВЕРШИНЫ ЗАДАЮТ ПЕРВЫЙ ОТРЕЗОК, СЛЕДУЮЩИЕ ДВЕ – ВТОРОЙ И Т.Д.</b></p>
<b>GL_LINE_STRIP</b>	<p>Задается ломаная. Каждая следующая вершина является концом текущего отрезка прямой и началом для следующего.</p> <p>Ошибка!</p> 
<b>GL_LINE_LOOP</b>	<p>Задается замкнутая ломаная. В отличие от предыдущего примитива (GL_LINE_STRIP) последний отрезок определяется последней и первой вершиной.</p> 

Продолжение табл.

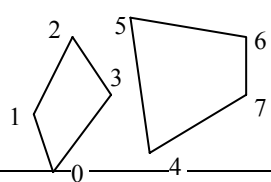
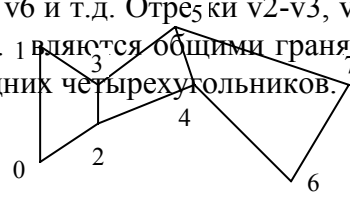

Значение <i>mode</i>	Описание
<b>GL_TRIANGLES</b>	<p>Задаются треугольники (треугольник). Каждые три вершины, сле-</p>



Значение <i>mode</i>	Описание
	<p>дующие друг за другом, определяют отдельный треугольник. Если количество вершин не кратно трем, то "лишние" вершины игнорируются.</p>
<b>GL_TRIANGLE_STRIP</b>	<p>Задается полоса из связанных треугольников, имеющих общую грань. Каждая вершина, начиная с третьей, вместе с двумя предыдущими задает треугольник. Следовательно первый треугольник – <math>v_0, v_1, v_2</math>; второй – <math>v_1, v_3, v_2</math>; третий – <math>v_2, v_4, v_3</math> и т.д. Отрезки <math>v_1-v_2, v_2-v_3, v_3-v_4</math> и т.д. являются общими гранями треугольников.</p>  <p><b>Ошибка!</b></p>
<b>GL_TRIANGLE_FAN</b>	<p>Задается "веер" связанных треугольников, имеющих общую вершину – первую вершину (<math>v_0</math>). Тогда первый треугольник – <math>v_0, v_1, v_2</math>; второй – <math>v_0, v_2, v_3</math>; третий – <math>v_0, v_3, v_4</math> и т.д. Отрезки <math>v_0-v_2, v_0-v_3, v_0-v_4</math> и т.д. являются общими гранями треугольников.</p>  <p><b>Ошибка!</b></p>

Окончание табл.

Значение <i>mode</i>	Описание
<b>GL_QUADS</b>	<p>Задаёт четырехугольник. Каждая последовательная четверка вершин определяет четырехугольник. Если количество вершин не кратно четырем, то "лишние" вершины</p>

Значение <i>mode</i>	Описание
	игнорируются. <b>Ошибка!</b> 
<b>GL_QUAD_STRIP</b>	Задаёт полосу из связанных четырёхугольников. Каждые два соседних четырёхугольника имеют одно общее ребро. Первый четырёхугольник – $v_0, v_1, v_3, v_2$ ; второй – $v_2, v_3, v_5, v_4$ ; третий – $v_4, v_5, v_7, v_6$ и т.д. Отрезки $v_2-v_3, v_4-v_5$ и т.д. являются общими гранями соседних четырёхугольников. 
<b>GL_POLYGON</b>	Задаётся выпуклый многоугольник. Последняя вершина автоматически соединяется с первой. 

Например, чтобы отобразить один четырёхугольник с разными цветами в вершинах, достаточно реализовать следующий фрагмент кода:

```
GLfloat Green[3] = {0.0, 1.0, 0.0};
glBegin(GL_QUADS);
glColor3f(1.0, 0.0, 0.0); // Красный
glVertex3f(0.0, 0.0, 0.0);
glColor3f(1.0, 1.0, 1.0); // Белый
glVertex3f(1.0, 0.0, 0.0);
glColor3fv(Green); // Зеленый
glVertex3f(1.0, 1.0, 0.0);
glColor3f(0.0, 0.0, 1.0); // Синий
glVertex3f(1.0, 1.0, 1.0);
glEnd();
```

В связи с тем, что на практике очень часто приходится рисовать прямоугольники, то в OpenGL включили команду рисования такого примитива:

```
void glRect[sifd] (type x1, type y1, type x2, type y2)
void glRect[sifd]v (const type *v1, const type *v2)
```

Эта команда рисует прямоугольник в плоскости  $z = 0$ . Левый нижний угол прямоугольника определяется параметрами  $(x1, y1)$ , а противоположный (по диагонали)  $(x2, y2)$ . В случае векторной формы команды, углы задаются в виде двух указателей на массивы, каждый из которых содержит значения  $(x, y)$ .

При визуализации многоугольников следует различать лицевую грань и нелицевую (обратную). Если при рассмотрении многоугольника его список вершин обходится по часовой стрелке, то такая грань многоугольника лицевая. Если против часовой, то – нелицевая.

В OpenGL существует механизм управления растеризацией многоугольников на экране. Для этого 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:

<i>face</i>	Вид грани
<b>GL_FRONT</b>	Лицевые
<b>GL_BACK</b>	Нелицевые
<b>GL_FRONT_AND_BAC</b>	Лицевые и нелицевые

<i>mode</i>	Вид отображения грани
<b>GL_POI NT</b>	Отображение только отдельных вершин многоугольников.
<b>GL_LINE</b>	Отображение многоугольников в виде набора отрезков.
<b>GL_FILL</b>	Многоугольники будут закрашиваться текущим цветом (режим установлен по умолчанию).

Для каждой вершины геометрического объекта можно задать свой вектор нормали следующей командой:

```
void glNormal3[b s i f d] ( type nx, type ny, type nz)
void glNormal3[b s i f d] v (const type *n)
```

Для включения автоматического приведения векторов к единичным применяют команду: `glEnable(GL_NORMALIZE)`.

### 5.3 Управление размерами точки

По умолчанию, размер точки, толщина линии в примитивах соответствует одному пикселю на экране, линия всегда непрерывная. Но эти параметры в OpenGL можно изменять. Рассмотрим некоторые команды предназначенные для этой цели.

Размер точки можно легко изменить командой – `void glPointSize(GLfloat size)`, где параметр *size* задает размер "квадрата" точки в пикселях. Дело в том, что при большом увеличении точки можно увидеть ее квадратность. Если данный параметр имеет значение 2, то точка будет выглядеть как квадрат 2×2. Для сглаживания углов "квадрата" точки больших размеров используется команда `glEnable(GL_POINT_SMOOTH)`, которая включает антиалиасинг точки. Выключение антиалиасинга для точки осуществляется командой `glDisable` с параметром `(GL_POINT_SMOOTH)`. По умолчанию антиалиасинг выключен.

Вот как выглядит фрагмент программы рисования "круглой" точки размером 3 пикселя:

```
glEnable(GL_POINTS); // Будем работать с точкой
glEnable(GL_POINT_SMOOTH); // Разрешаем антиалиасинг точки
glPointSize(3.0); // Задаем размер точки
glBegin(GL_POINTS); // Определяем координаты рисования
glVertex2i(100,100);
glEnd();
```

```
glDisable(GL_POINT_SMOOTH); // Запрещает антиалиасинг точки
```

#### 5.4 Управление толщиной и шаблоном линии

По умолчанию, толщина линии в примитивах соответствует одному пикселю на экране, линия всегда непрерывная. Эти параметры в OpenGL можно изменять. Рассмотрим некоторые команды предназначенные для этой цели.

Толщина линии задается командой `void glLineWidth (GLfloat width)`

Параметр *width* задает в пикселях толщину линии и должен быть больше (0.0f).

Как и в случае с точкой, мы можем включать или выключать сглаживание линии командами `glEnable/glDisable` с параметром `GL_LINE_SMOOTH`. **Внимание!** Если не включать сглаживание линии, а этот режим является базовым по умолчанию, то толщина линии будет визуально изменяться в зависимости от угла наклона.

Дело в том, что в этом режиме толщина линии определяется вдоль оси *y*, если  $|y_1 - y_2| < |x_1 - x_2|$  (где  $(x_1, y_1)$  и  $(x_2, y_2)$  – координаты концов отрезка) и в направлении оси *x* в ситуации  $|y_1 - y_2| > |x_1 - x_2|$ . А в случае сглаживания толщина линии замеряется перпендикулярно линии.

Если требуется рисовать линию фрагментарно, т.е. не непрерывно, то для этой цели используют шаблоны. Шаблон линии задается командой:

```
void glLineStipple (GLint factor, GLushort pattern)
```

Параметр *pattern* – это и есть шаблон 16-точечного фрагмента линии, который состоит из серии нулей и единиц. Единица означает, что соответствующая точка линии будет нарисована на экране, ноль означает, что точка нарисована не будет. Запись шаблона начинается с младшего бита параметра *pattern*. Например, шаблон вида `0x00FF` запретит выводить первые восемь точек отрезка линии, после чего будет разрешено выводить восемь точек, и так до конца линии. К шаблону можно применить масштабирование (в сторону увеличения), это определяется параметром *factor*. Если *factor* равен 2, то каждый бит шаблона при наложении на отрезок будет представлен двумя точками идущих друг за другом. Например, если в шаблоне встречаются подряд две единицы, а затем три нуля и *factor* равен 2, то шаблон этот фрагмент рассматривается как содержащий 4 единицы и 6 нулей. Максимальное увеличение шаблона ограничено числом 256. Использование команды `glLineStipple()` должно быть разрешено командой `glEnable()` с параметром `GL_LINE_STIPPLE`. Команда `glDisable()` с этим же параметром запрещает работу с шаблоном.

Рассмотрим пример построения линии состоящей из повторяющихся точек через одну. Для этого потребуется шаблон `0xAAAA`:

```
glLineStipple(1,0xAAAA); // Задаем шаблон с масштабом 1
glEnable(GL_LINE_STIPPLE); // Разрешаем работу шаблоном
glEnable(GL_LINES);
glBegin(GL_LINES);
glColor3f(1.0f,1.0f,0.0f);
glVertex2i(40,120);
glVertex2i(240,120);
glEnd();
glDisable(GL_LINES);
glDisable(GL_LINES_STIPPLE); // Запрещаем работу с шаблоном
```

Если в приведенном примере в команде `glLineStipple(1,0xAAAA)` заменить масштабный множитель шаблона 1 на значение 20 (`glLineStipple(20,0xAAAA)`), то на экране вместо точек увидим отрезки прямых длиной в 20 пиксель.

#### 5.5 Грани плоских многоугольников

Каждый плоский многоугольник имеет две стороны – лицевую и обратную. Если посмотреть на поверхность многоугольника и описать его списком вершин обходя многоугольник против хода часовой стрелки, то такая грань называется лицевая. В противном случае – обратная. Очевидно, что лишь одна грань многоугольника может быть видна. Чтобы не делать лишней работы на невидимых гранях объекта необходимо эту ситуацию отслеживать. По умолчанию в OpenGL считается, что многоугольник задается вершинами против часовой стрелки. Но это правило можно изменить командой

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

Значением параметра *mode* можно изменять правило задания вершин для лицевых граней. Если этот

параметр принимает значение символьной константы **GL\_CW** то признак лицевой грани – обход вершин по часовой стрелке и **GL\_CCW** – против часовой стрелки.

Лицевые или обратные грани можно целенаправленно исключать из процесса вывода полигонов. Рассмотрим команду осуществляющую управление выводом полигонов в зависимости от типа грани:

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

Если значение параметра *mode* равно **GL\_FRONT**, то не будут отображаться лицевые грани. Также возможны значения **GL\_BACK** и **GL\_FRONT\_AND\_BACK**. По умолчанию берется значение **GL\_BACK**, т.е. выводить только лицевые грани. Перед тем как использовать **glCullFace()** необходимо выполнить команду **glEnable(GL\_CULL\_FACE)**, которая разрешит использовать данный режим. Блокировка данного режима осуществляется командой **glDisable(GL\_CULL\_FACE)**.

Рассмотрим небольшой пример работы с лицевыми и обратными гранями полигонов:

```
...
glEnable(GL_CULL_FACE); // Вкл. режим упр. выводом граней
//glCullFace(GL_FRONT); // Исключить вывод лицевых граней
glColor3f(1.0f, 0.0f, 0.0f); //Красный цвет для треугольника
glBegin(GL_TRIANGLES); // Треугольник (против часовой)
glVertex3i( 100, 100, 0); // Вершина 1
glVertex3i( 50, 50, 0); // Вершина 2
glVertex3i( 150, 50, 0); // Вершина 3
glEnd();
glColor3f(0.0f, 1.0f, 0.0f); //Зеленый цвет для прямоугольника
glBegin(GL_QUADS); // Прямоугольник (по часовой)
glVertex3i(200, 100, 0); // Вершина 1
glVertex3i(250, 100, 0); // Вершина 2
glVertex3i(250, 50, 0); // Вершина 3
glVertex3i(200, 50, 0); // Вершина 4
glEnd();
...
```

Если посмотреть на результат работы данного фрагмента программы, то можем увидеть на экране только красный треугольник, так как зеленый прямоугольник был отброшен (по умолчанию в данном режиме отбрасываются обратные грани полигонов). Если разблокировать команду **glCullFace(GL\_FRONT)**(убрать комментарии), то на экране вместо треугольника появится прямоугольник зеленого цвета. Его описание соответствует нелицевой грани.

## 5.6 Заполнение полигонов шаблонами

По умолчанию полигоны заполняются без рисунка – однородным значением. Для заполнения полигона повторяющимся орнаментом используют команду, использующую битовую карту:

```
void glPolygonStipple (const GLubyte *mask)
```

Предварительно готовится битовая карта размером 32×32 бит, являющаяся маской, циклично (по мере необходимости) накладываемой на полигон при его выводе в растр. Там, где стоит 1 – соответствующий пиксель полигона будет нарисован, а там, где появляется 0 – пиксель нарисован не будет. Параметр *mask* – это указатель на массив битовой карты. Покрытие шаблоном полигона разрешается и запрещается с помощью команд **glEnable()** и **glDisable()** с параметром **GL\_POLYGON\_STIPPLE**. Варьируя параметрами **GL\_UNPACK\*** команды **glPixelStore\*()** можно изменять режим хранения пикселей в памяти (смотри MSDN), что повлечет за собой видоизменение внутреннего рисунка в полигоне.

Рассмотрим пример использования шаблона для заполнения четырехугольного полигона. Первые четыре байта (0xff,0xff,0xff,0xff) массива **Bit\_Map** описывают нижнюю строчку трафарета, вторая четверка (0x01,0x01,0x01,0x01) – вторую снизу строчку трафарета и т.д. Более наглядней оформлять листинг шаблона в программе матрицей размером 4x32, но для экономии места приходится 16x16, т.е. две строки шаблона записывать в одной строке листинга.

```
GLvoid DrawPrim()
{
// Формируем шаблон полигона 32x32 бит!!!
```

```

GLubyte Bit_Map[] = {
0xff,0xff,0xff,0xff,0x01,0x01,0x01,0x01,
0x00,0x00,0x00,0x00,0x01,0x01,0x01,0x01,
... (для экономии места - пропущены шесть одинаковых наборов)
0xff,0xff,0xff,0xff,0x01,0x01,0x01,0x01,
0x00,0x00,0x00,0x00,0x01,0x01,0x01,0x01 };
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,1.0,1.0); //Задаем белый цвет
glEnable(GL_POLYGON_STIPPLE); //Включаем шаблон
glPolygonStipple(Bit_Map); //Определяем шаблон
glBegin(GL_POLYGON); //Рисуем полигон
glVertex2i(10, 10);
glVertex2i(30, 60);
glVertex2i(100, 70);
glVertex2i(80, 20);
glEnd();
glDisable(GL_POLYGON_STIPPLE); //Выключаем шаблон
}

```

### 5.7 Воспроизведение битовых массивов

В OpenGL имеется возможность рисовать растровые изображения на основании одномерного битового массива. Там, где стоит 1 – пиксель будет нарисован, а там, где появляется 0 – пиксель нарисован не будет. Битовый массив при выводе разбивается на строки, которые, в свою очередь, набираются в высоту изображения. Для этого используется команда

```

void glBitmap (GLsizei width, GLsizei height,
              GLfloat xorig, GLfloat yorig,
              GLfloat xmove, GLfloat ymove,
              const GLubyte *bitmap)

```

Параметры width и height задают в пикселях ширину и высоту точечного рисунка, представленного в виде битового массива, xorig и yorig уточняют адрес левого нижнего угла точечного изображения в плоскости экрана по формулам  $x_{\text{окн}} = x_{\text{тек}} - x_{\text{orig}}$  и  $y_{\text{окн}} = y_{\text{тек}} - y_{\text{orig}}$ , где  $x_{\text{окн}}$  и  $y_{\text{окн}}$  оконные координаты,  $x_{\text{тек}}$  и  $y_{\text{тек}}$  текущие координаты растровой позиции, задаваемые командой `glRasterPos*()`. Параметры xmov и ymov показывают на сколько изменится текущая позиция раstra после вывода битового изображения. Переменная bitmap указывает на адрес в оперативной памяти компьютера, начиная с которого хранится битовый образ.

Перед тем, как рассмотреть несколько примеров использования команды `glBitmap()` необходимо разобраться как в OpenGL задаются координаты текущей точки раstra. Для установки оконных координат используется команда

```

void glRasterPos[234][s I f d](GLtype coords)
void glRasterPos[234][s I f d]v(GLtype coords)

```

где coords – координаты текущей позиции раstra.

По умолчанию текущая позиция раstra установлена по координатам (0, 0, 0, 1).

Кроме этого, для успешной работы с командой `glBitmap()` требуется устанавливать режим хранения пиксель в памяти. А это нам обеспечит команда

```

void glPixelStore[f i](GLenum pname, GLtype param)

```

Параметр pname задает символьную константу, указывающую на определенный устанавливаемый параметр, а param – конкретное значение для данного параметра. Рассмотрим возможные символьные константы для параметра pname:

<i>Pname</i>	Значение <i>param</i>
--------------	-----------------------

<i>Pname</i>	Значение <i>param</i>
GL_PACK_SWAP_BYTES, GL_UNPACK_SWAP_BYTES	• ЕСЛИ TRUE, ТО БАЙТЫ В МНОГОБАЙТНЫХ КОМПОНЕНТАХ ЦВЕТА, ГЛУБИНЫ И ИНДЕКСА ТРАФАРЕТА УПОРЯДОЧЕНЫ В ОБРАТНОМ ПОРЯДКЕ
GL_PACK_LSB_FIRST, GL_UNPACK_LSB_FIRST	• Если TRUE, то биты внутри байта упорядочены от младших разрядов к старшим
GL_PACK_ROW_LENGTH, GL_UNPACK_ROW_LENGTH	• <i>param</i> определяет число пикселей в строке
GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS	• Значением <i>param</i> можно задавать количество пропускаемых пиксель или строк
GL_PACK_ALIGNMENT, GL_UNPACK_ALIGNMENT	• Выравнивание начала строки пикселей в памяти: 1 – по байту; 2 – по четному байту; 4 – по слову; 8 – по двойному слову. По умолчанию – 4 (по слову)

Теперь мы можем рассмотреть пример работы команды `glRasterPos*()`. Пусть требуется вывести на экран в точке с координатами (100, 50) символ Т. Образ его и соответствующие кодировочные числа приведены на рисунке 1.

Рассмотрим эту задачу по шагам.

**1 шаг:** Кодировочные числа разместим в оперативной памяти.

```
GLubyte BitMap[24] = {
0x0c,0x00,0x0c,0x00,0x0c,0x00,0x0c,0x00,0x0c,0x00,0x0c,0x00,0x0c,0x00,
0x0c,0x00,0x0c,0x00,0x0c,0x00,0x0c,0x00,0xff,0xc0,0xff,0xc0};
```

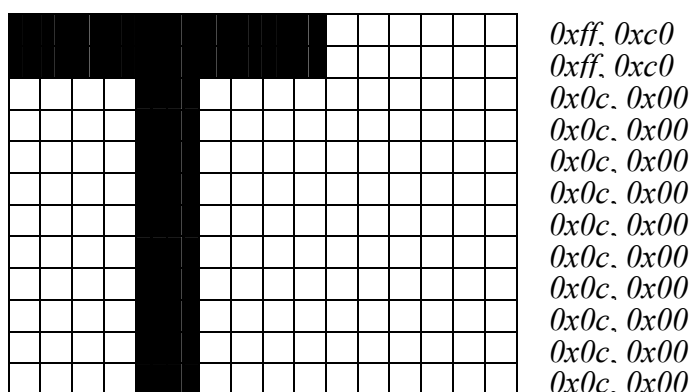


Рис. 1 Битовая матрица символа Т

**Внимание!** Битовый образ записывается в массив снизу вверх и справа налево.

**2 шаг:** Указываем OpenGL, что пиксели битового образа в памяти будут выровнены по байту:



```
glPixelStorei (GL_UNPACK_ALIGNMENT, 1);
3 шаг: Определяем текущую позицию на экране:
glRasterPos2i (120, 120);
4 шаг: Рисуем на экране битовый образ:
glBitmap (16, 12, 0.0, 0.0, 0.0, 0.0, BitMap);
```

По этой команде будет выведен точечный рисунок, взятый по адресу BitMap, размером 16×12 и без уточнения координат точки привязки, так как третий и четвертый параметр – (0.0, 0.0). Судя по пятому и шестому параметру (0.0, 0.0) текущая точка на экране после вывода битового образа не изменит своих координат.

Для изучения пятого и шестого параметров команды glBitmap() рассмотрим еще один пример с этим же символом. Допустим, теперь необходимо сформировать строку из трех одинаковых символов. Код программы этой операции имеет вид:

```
glClear(GL_COLOR_BUFFER_BIT); // Очищаем буфер цвета
glColor3f (1.0, 1.0, 1.0); // Выбираем белый цвет
glRasterPos2i (120, 120); // Текущая позиция экрана
glBitmap (16, 12, 0.0, 0.0, 16.0, 0.0, BitMap); //1 символ
glBitmap (16, 12, 0.0, 0.0, 16.0, 0.0, BitMap); //2 символ
glBitmap (16, 12, 0.0, 0.0, 16.0, 0.0, BitMap); //3 символ
```

Если потребуется вывести в строке разные символы, то для этого формируются несколько массивов точечных рисунков символов, например BitMap\_A, BitMap\_B и т.д. Следующий фрагмент программы выводит символы A, B и C в одну строчку.

```
glBitmap (16, 12, 0.0, 0.0, 16.0, 0.0, BitMap_A);
glBitmap (16, 12, 0.0, 0.0, 16.0, 0.0, BitMap_B);
glBitmap (16, 12, 0.0, 0.0, 16.0, 0.0, BitMap_C).
```

Расстояние между битовыми образами символов будет равно нулю, так как последующая позиция символа отличается от предыдущей на 16 (пятый параметр), что соответствует длине битового образа. Но на экране расстояние между символами мы будем все же наблюдать, лишь только потому, что при кодировке символа межсимвольное расстояние зарезервировано в его битовом образе (справа от символа 6 столбцов) (см. рис. 1).

## 5.8 Отсечение областью экрана

Часто на практике требуется вывести графический примитив в рамках определенной прямоугольной области экрана. Если часть объекта выходит за рамки этой области, то она не выводится на экран. Эту задачу решает тест отсечения, задаваемый командой

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height)
```

Эта команда задает прямоугольник отсечения:  $x$  и  $y$  – оконные координаты левой нижней угол области отсечения,  $width$  и  $height$  – ширина и высота этой области. Все параметры задаются в пикселях. В первый раз, когда контекст воспроизведения подсоединяется к окну,  $x$  и  $y$  принимают значения (0, 0), а  $width$  и  $height$  – его ширину и высоту.

Для разрешения и запрещения отсечения используются команды glEnable() и glDisable() с параметром **GL\_SCISSOR\_TEST**.

Если выполнить команду glScissor(0, 0, 0, 0), то будет заблокирован вывод графики на экран, так как любая выводимая точка будет выходить за рамки отведенной области.

## 5.9 Отсечение объектов плоскостями

OpenGL предоставляет возможность разработчикам обрезать графические 3D объекты. В качестве секущей плоскости используется плоскость, заданная коэффициентами уравнения вида  $Ax + By + Cz + D = 0$ . Напомним, вектор  $\{A, B, C\}$  – задает нормаль к плоскости. Для отсечения используется команда

```
void glClipPlane (GLenum plane, const GLdouble *equation)
```

Параметр *plane* определяет одну из шести (точнее – **GL\_MAX\_CLIP\_PLANES**) секущих плоскостей и может принимать следующие символьные значения: **GL\_CLIP\_PLANE $i$** , где  $i$  – целое число от 0 до 5 (**GL\_MAX\_CLIP\_PLANES**-1). Параметр *equation* – указатель на массив из четырех значений (A, B, C и D), задающих уравнение плоскости. **Внимание:** Данный вектор указывает своим направлением на

оставшуюся часть объекта после отсечения. Для разрешения работы с подобными плоскостями необходимо выполнить команду `glEnable()` с аргументом `GL_CLIP_PLANEi`. Отключение текущей плоскости происходит после команды `glDisable()` с соответствующим аргументом.

Рассмотрим пример отсечения некоторого объекта двумя плоскостями. Первая плоскость отсекает полупространство для которого  $x < 1$  и вторая плоскость отсекает полупространство для которого  $y < 0$

```

...
// Коэффициенты A, B, C и D для первой плоскости отсечения
GLdouble P10[] = {1.0, 0.0, 0.0, 1.0};
// Коэффициенты A, B, C и D для второй плоскости отсечения
GLdouble P11[] = {0.0, 1.0, 0.0, 0.0};
// Идентифицируем первую плоскость отсечения
glClipPlane(GL_CLIP_PLANE0, P10);
// Идентифицируем вторую плоскость отсечения
glClipPlane(GL_CLIP_PLANE1, P11);
// Включаем отсечение первой плоскостью
glEnable(GL_CLIP_PLANE0);
// Включаем отсечение второй плоскостью
glEnable(GL_CLIP_PLANE1);
// Здесь выводим объект ...

```

Следует помнить, что параметры текущей плоскости автоматически изменяются в соответствии с модельными и видовыми преобразованиями. И еще, вектор нормали (A, B, C) не должен быть обязательно перпендикулярен осям, как в нашем примере.

### 5.10 Логические операции при выводе объекта

При поступлении значений объекта в буфер кадра в OpenGL обозначаются несколько интересных ситуаций: либо эти значения просто замещают существующие, либо происходит логическая операция между данными, которые находятся в соответствующем месте буфера кадра с поступающими данными. Задание логической операции определяется командой `void glLogicOp(GLenum opcode)`.

Параметр *opcode* задает логическую операцию и может принимать следующие значения (принятые обозначения: *s* – значение бита источника, *d* – значение бита приемника).

<i>Opcode</i>	Логическая операция
<code>GL_CLEAR</code>	0
<code>GL_SET</code>	1
<code>GL_COPY</code>	s
<code>GL_COPY_INVERTED</code>	!s
<code>GL_NOOP</code>	d
<code>GL_INVERT</code>	!d
<code>GL_AND</code>	s&d
<code>GL_NAND</code>	!(s&d)
<code>GL_OR</code>	s d
<code>GL_NOR</code>	!(s d)
<code>GL_XOR</code>	s^d
<code>GL_EQUIV</code>	!(s^d)
<code>GL_AND_REVERSE</code>	s&!d
<code>GL_AND_INVERTED</code>	!s&d
<code>GL_OR_REVERSE</code>	s !d
<code>GL_OR_INVERTED</code>	!s d

Для разрешения подобного режима работы необходимо обработать команду `glEnable()` с аргументом `GL_COLOR_LOGIC_OP` для полноцветного цветового режима. Выключается режим командой `glDis-`

able() с таким же аргументом. Напомним, логическая операция выполняется побитно над данными.

## 6 РИСОВАНИЕ ГОТОВЫХ 3D ОБЪЕКТОВ

Рассмотренные выше графические примитивы являются базовыми элементами для построения графических замыслов. Но в библиотеке GLU, которая реализована в виде модуля glu32.dll, описаны более сложные графические объекты, такие как сфера, цилиндр, диск и часть диска.

Для построения подобных примитивов необходимо создать указатель на quadric-объект с помощью команды gluNewQuadric(), а после рисования удалить вызовом функции gluDeleteQuadric. Схема рисования выглядит следующим образом:

```
GLUquadricObj *quadric_obj;  
quadric_obj=gluNewQuadric();
```

```
...
```

```
// вызов команды рисования графического объекта GLU  
gluDeleteQuadric(quadric_obj).
```

Рассмотрим команды рисования графических объектов отдельно:

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

Данная команда строит сферу с центром в начале координат и радиусом *radius*. Число разбиений сферы вокруг оси *z* задается параметром *slices*, а вдоль оси *z* параметром *stacks*. **Внимание!** Интересный эффект можно получить от сферы если задавать значения последних двух параметров из ряда величин: 2, 3, 4 и т.д.

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

Эта команда строит цилиндр без оснований (кольцо), продольная ось параллельна оси *z*, заднее основание имеет радиус *baseRadius*, и расположено в плоскости  $z = 0$ , переднее основание имеет радиус *topRadius* и расположено в плоскости  $z = height$ . **Внимание!** Если задать один из радиусов равным нулю, то будет построен конус. Параметры *slices* и *stacks* имеют тот же смысл, что и в команде gluSphere. **Внимание!** при увеличении значения *stacks* (больше двух) улучшается эффект сглаживания интерполяционной закраски поверхности.

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

Данная команда строит плоский диск (т.е. круг) с центром в начале координат и радиусом *outerRadius*. Если значение *innerRadius* ненулевое, то в центре диска будет находиться отверстие радиусом *innerRadius*. Параметр *slices* задает число разбиений диска вокруг оси *z*, а параметр *loops* число концентрических окружностей, перпендикулярных оси *z*.

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

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

Кроме этих готовых графических объектов в OpenGL можно использовать объекты предоставленные библиотекой GLUT: сфера, куб, конус, тор, тетраэдр, додекаэдр, икосаэдр, октаэдр и чайник. Для детального изучения команд моделирования подобных объектов рекомендуем обратиться к соответствующей документации.

## 7 РАБОТА С МАССИВАМИ ВЕРШИН И ИХ АТТРИБУТОВ

При моделировании объектов с большим количеством вершин можно не вызывать многократно команду `glVertex*()`, а воспользоваться командой:

```
void glVertexPointer (GLint s, GLenum t, GLsizei st, const GLvoid *p)
```

Данная команда задает массив с координатами вершин. Здесь  $s$  – количество координат в каждой вершине и должно быть установлено в 2, 3 или 4. Параметр  $t$  задает тип значения для координаты в массиве и может принимать одну из следующих символьных констант `GL_SHORT`, `GL_INT`, `GL_FLOAT` и `GL_DOUBLE`. Параметр  $st$  принимает значение смещения в байтах между координатами соседних последовательных вершин. Если  $st$  равно нулю, то это означает, что координаты вершин расположены последовательно в массиве. Иногда удобно в массиве располагать не только значения координат вершин, но и некоторые сопутствующие атрибуты (текстурные координаты, нормали вершин, значения цвета и т.д.). В этом случае параметр  $st$  позволяет "раздвинуть" координаты вершин в массиве и оставить места для дополнительных данных. Параметр  $p$  является указателем на координаты первой вершины в массиве.

Такой подход к организации вершин объекта позволяет повысить скорость передачи данных в OpenGL, что, в конечном счете, положительно влияет на скорость рисования этих объектов на экране.

Аналогично можно организовать массивы нормалей, цветов и текстурных координат. Для это существуют соответствующие команды:

```
void glNormalPointer (GLenum t, GLsizei s, const GLvoid *p)
```

```
void glColorPointer (GLint s, GLenum t, GLsizei st, const GLvoid *p)
```

```
void glTexCoordPointer (GLint s, GLenum t, GLsizei st, const GLvoid *p)
```

Чтобы разрешить работу с подобными массивами, необходимо вызвать команду `glEnableClientState()` с соответствующим аргументом:

Аргумент	Назначение
<code>GL_VERTEX_ARRAY</code>	ДЛЯ МАССИВА ВЕРШИН
<code>GL_NORMAL_ARRAY</code>	Для массива нормалей
<code>GL_COLOR_ARRAY</code>	Для массива цветов
<code>GL_TEXTURE_COORD_ARRAY</code>	Для текстовых координат

Для запрещения работы с массивом используется команда `glDisableClientState` с соответствующим аргументом.

После того как требуемые массивы сформированы и работа с ними разрешена, воспроизведение примитива осуществляется вызовом команды

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

Эта команда осуществляет воспроизведение примитива, заданного параметром  $mode$  (см. команду `glBegin/glEnd`). Всего отображается  $count$  элементов, начиная с номера, указанного параметром  $first$ .

Для отображения единственного параметра примитива используется команда

```
void glVertexAttribPointer (GLint index)
```

Параметр  $index$  задает номер отображаемого элемента примитива.

Кстати, внутренний механизм работы команды `glDrawArrays()` основан на последовательном выполнении команды `glVertexAttribPointer()`  $count$  раз, начиная с  $first$  примитива.

Для закрепления материала рассмотрим простой пример. Пусть требуется нарисовать с помощью `glDrawArrays()` пятиугольник с различными цветами вершин. Рассмотрим выполнение этой задачи по шагам:

**1 шаг:** Объявляем массивы вершин ( $v$ ) и цвета ( $c$ ).

```
GLfloat V[5][2]; // Для x и y
```

```
GLfloat C[5][3]; // Для R, G и B
```

**2 шаг:** Заполняем массив ( $v$ ) исходными значениями координат вершин пятиугольника и массив ( $c$ )

атрибутами цвета.

```
V[0][0] = 0.0; V[0][1] = 2.0; C[0][0] = 1.0; C[0][1] = 0.0; C[0][2] = 0.0;
V[1][0] = 2.0; V[1][1] = 1.0; C[1][0] = 0.0; C[1][1] = 1.0; C[1][2] = 0.0;
V[2][0] = 1.5; V[2][1] = -1.0; C[2][0] = 0.0; C[2][1] = 0.0; C[2][2] = 1.0;
V[3][0] = -1.5; V[3][1] = -1.0; C[3][0] = 0.0; C[3][1] = 0.5; C[3][2] = 0.0;
V[4][0] = -2.0; V[4][1] = 1.0; C[4][0] = 1.0; C[4][1] = 0.5; C[4][2] = 1.0;
```

**3 шаг:** Указываем OpenGL, что данные этих массивов будут использоваться командами `glVertexPointer` и `glColorPointer` для размещения в специализированных внутренних массивах вершин и цветов.

```
glVertexPointer(2, GL_FLOAT, 0, V);
```

```
glColorPointer(3, GL_FLOAT, 0, C);
```

**4 шаг:** Разрешаем работать OpenGL с внутренним массивом вершин и массивом цветов.

```
glEnableClientState(GL_VERTEX_ARRAY);
```

```
glEnableClientState(GL_COLOR_ARRAY);
```

**5 шаг:** Рисуем пятиугольник.

```
glDrawArrays(GL_POLYGON, 0, 5);
```

**6 шаг:** Делаем недоступными массивы вершин и цветов

```
glDisableClientState(GL_VERTEX_ARRAY);
```

```
glDisableClientState(GL_COLOR_ARRAY);
```

Вот и все.

Если воспользоваться третьим параметром команды `gl*Pointer`, то подобную задачу можно выполнить более компактно. Для этого разместим исходные данные на вершины и их цветов в перебежку (вершина, цвет, вершина, цвет т.д.) в одном одномерном массиве:

```
GLfloat VC[] = { 0.0, 2.0, 1.0, 0.0, 0.0,
                2.0, 1.0, 0.0, 1.0, 0.0,
                1.5, -1.0, 0.0, 0.0, 1.0,
                -1.5, -1.0, 0.0, 0.5, 0.0,
                -2.0, 1.0, 1.0, 0.5, 1.0 };
```

Далее вызываем команды `glVertexPointer` и `glColorPointer`:

```
glVertexPointer(2, GL_FLOAT, 5*sizeof(GLfloat), &VC[0]);
```

```
glColorPointer(3, GL_FLOAT, 5*sizeof(GLfloat), &VC[2]);
```

Обратите внимание на то, что массив вершин будет набираться с первого элемента массива `VC` по два значения типа `float` с шагом `5*sizeof(GLfloat)`. На правильность выбранного шага подсказывает тот факт, что порядковые номера одноименных данных в массиве `VC` отличаются друг от друга на 5. А так как тип элементов массива `float`, то и смещение определяется выражением `5*sizeof(GLfloat)`. Первые два числа в массиве всегда относятся к вершине объекта, а начиная с третьего, точнее с номера 2, так как нумерация элементов в массиве идет с номера `-0`, параметры цвета соответствующей вершины.

И в завершении уже знакомые команды:

```
glDrawArrays(GL_POLYGON, 0, 5);
```

```
glDisableClientState(GL_VERTEX_ARRAY);
```

```
glDisableClientState(GL_COLOR_ARRAY);
```

На экране мы получаем такой же результат, что и в предыдущем примере.

## 8 РАБОТА СО СПИСКАМИ ИЗОБРАЖЕНИЙ

В OpenGL предоставляется возможность объединять группу команд под определенным именем для последующего выполнения. Это обеспечивают списки изображений (дисплейные списки). Они подобны подпрограммам и являются удобным средством для кодирования больших систем со сложной иерархией. Список организуется командами:

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

```
void glEndList ()
```

Эти команды выглядят, как операторные скобки. Первая команда является заголовком (началом) нового списка изображений. Список заканчивается командой `glEndList()`. Параметр `list` представляет собой целочисленный идентификатор – имя списка. Параметр `mode` – символическая константа, определяющая следующие режимы сборки списка:

Константа	Режим сборки списка
-----------	---------------------

<b>GL_COMPILE</b>	<b>СПИСОК КОМАНД ТОЛЬКО КОМПИЛИРУЕТСЯ (НАКАПЛИВАЕТСЯ) ДЛЯ ВОЗМОЖНОГО ПОСЛЕДУЮЩЕГО ИСПОЛЬЗОВАНИЯ</b>
<b>GL_COMPILE_AND_EXECUTE</b>	Команды исполняются и только потом заносятся в список

OpenGL компилирует список изображений только один раз, после этого он готов к применению. Поэтому мы получаем увеличение скорости при использовании списков изображений.

Порядок, в котором команды располагаются в списке изображений, жестко определяют порядок, в котором они будут выполняться.

Вызов списка изображений осуществляется командой:

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

Параметр *list* задает целочисленное имя списка изображений.

Рассмотрим пример подготовки и вызова списка изображений.

```
...
// Открываем список изображений (без выполнения) под номером 2
glNewList(2, GL_COMPILE);
glColor3f(1.0f, 0.0f, 0.0f); // Задаем цвет создаваемого примитива
glBegin(GL_POLYGON); // Задаем сам примитив
glVertex3f( 1.0f, 1.0f, 0.0f);
glVertex3f(-1.0f, 1.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, 0.0f);
glEnd();
glEndList(); // Закрываем список изображений под номером 2
...
glCallList(2);
...
```

В список можно включать вызовы уже готовых списков изображений. Рассмотрим список из двух вызовов готовых списков 1 и 2:

```
...
// Открываем список изображений (без выполнения) под номером 1
glNewList(1, GL_COMPILE);
glColor3f(0.0f, 1.0f, 0.0f);
glBegin(GL_TRIANGLES);
glVertex3f( 1.0f, 2.0f, 0.0f);
glVertex3f(-1.0f, 2.0f, 0.0f);
glVertex3f(-1.0f, -2.0f, 0.0f);
glEnd();
glEndList();
// Открываем список изображений (без выполнения) под номером 2
glNewList(2, GL_COMPILE);
glColor3f(1.0f, 0.0f, 0.0f);
glBegin(GL_TRIANGLES);
glVertex3f( 1.0f, 1.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, 0.0f);
glVertex3f(1.0f, -1.0f, 0.0f);
glEnd();
glEndList();
// Открываем список изображений (без выполнения) под номером 3
glNewList(3, GL_COMPILE);
glCallList(1);
glCallList(2);
```

```
glEndList();
```

```
...
```

```
// Вызов списка 3
```

```
glCallList(3);
```

```
...
```

Для выполнения сразу нескольких готовых списков используется команда:

```
void glCallList (GLsizei n,
```

```
GLenum type,
```

```
Const GLvoid *lists)
```

где  $n$  – количество исполняемых списков,  $lists$  – указатель на массив, содержащий список имен списков изображений, которые необходимо исполнить,  $type$  – тип значений в списке имен  $lists$  и может принимать следующие символьные константы:

Константа	Тип значений в списке имен изображений
<b>GL_BYTE</b>	<b>ПАРАМЕТР <i>LISTS</i> ЯВЛЯЕТСЯ УКАЗАТЕЛЕМ НА МАССИВ С БАЙТОВЫМИ ЭЛЕМЕНТАМИ, ПРИНИМАЮЩИМИ ЗНАЧЕНИЯ В ДИАПАЗОНЕ [–128, 127]</b>
<b>GL_UNSIGNED_BYTE</b>	Параметр $lists$ является указателем на массив с положительными байтовыми элементами, принимающими значения в диапазоне [0, 255]
<b>GL_SHORT</b>	Параметр $lists$ является указателем на массив с двухбайтовыми элементами, принимающими значения в диапазоне [–32768, 32767]
<b>GL_UNSIGNED_SHORT</b>	Параметр $lists$ является указателем на массив с положительными двухбайтовыми элементами, принимающими значения в диапазоне [0, 65535]
<b>GL_INT</b>	Параметр $lists$ является указателем на массив с четырехбайтовыми целыми элементами
и т.д. (см. MSDN)	

Перед вызовом команды `glCallLists` можно задать постоянное смещение к каждому имени списка изображения из массива  $lists$ . Это выполняется командой

```
void glListBase (GLuint base)
```

где  $base$  – целочисленное смещение. По умолчанию это смещение равно нулю.

Как уже было показано в примере, команды `glCallList` и `glCallLists` могут быть вложены в другие списки изображений. Уровней вложения не может быть больше 64.

При работе со списками, особенно при формировании шрифтов, приходится следить за номерами создаваемых списков, что не всегда удобно. Эту проблему легко разрешает функция `glGenList`, генерирующая непрерывный набор пустых дисплейных списков. Рассмотрим формат записи этой команды.

```
GLuint glGenList (GLsizei range)
```

Параметр  $range$  определяет количество пустых дисплейных списков, которые необходимо сгенерировать. Функция `glGenList` возвращает целое число  $n$ , которое является именем первого дисплейного списка. Если  $range$  больше 1, то также будут созданы пустые дисплейные списки с именами  $n+1$ ,  $n+2$ , ...,  $n+range-1$ . При нулевом значении  $range$  функция `glGenList` вернет значение 0 и никакой дисплейный список не создастся.

В ходе выполнения программы можно удалять непрерывную группу имен списков изображений. Допустим, имеется группа имен, например: 1, 3, 4, 5, 6, 8, 9, 10. Мы можем удалить из этой группы любую непрерывную последовательность имен списков, например, 4, 5 и 6. Для этого необходимо воспользоваться специальной командой

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

где *list* – целое число, являющееся именем первого удаляемого списка изображений, *range* – количество удаляемых списков.

Для приведенного выше примера, оформление команды `glDeleteLists` выглядит следующим образом: `glDeleteLists(4,3)`.

## 9 ВЫВОД ТЕКСТА

Текст в OpenGL выводится на экран посимвольно и каждый символ представляется как обычный графический образ. Для этого, предварительно, каждый символ, выбранного шрифта, готовится в виде `BitMap` (битового массива) в своем индивидуальном дисплейном списке. Так как дисплейный список имеет числовое имя, то зная код выводимого символа и соответствующее базовое смещение по отношению названий (номеров) дисплейных списков мы можем активизировать соответствующий список изображения. Рассмотрим эту концептуальную схему вывода текста на экран подробно.

Графическая библиотека имеет готовую команду, строящую набор дисплейных списков для символов заданного шрифта типа `TrueType` – команду

```
BOOL wglUseFontBitmaps (HDC hdc, DWORD first,  
                        DWORD count, DWORD listBase)
```

Параметр *hdc* – ссылка на контекст устройства, шрифт которого будет использоваться, параметр *first* – первый код символа, *count* – количество символов и *listBase* – стартовый дисплейный список. В итоге после отработки этой команды будут сформированы *count* битовых образов символов шрифта устройства *hdc*, начиная с кода *first*. И каждый битовый образ будет размещен в своем дисплейном списке, начиная с номера *listBase*. Для указания конкретного места вывода на экране используется команда `glRasterPos2i()`. Цвет будущего текста настраивается командой `glColor*()`. И так, пример вывода текста – "Привет OpenGL !!!" :

```
...  
// Выбираем текущий шрифт системы  
SelectObject (hDC, GetStockObject (SYSTEM_FONT));  
// Форм. 255 дисплейных списков изображений символов, начиная с  
// кода 1 и размещая с базового имени дисплейного списка – 1  
wglUseFontBitmaps (hDC, 1, 255, 1);  
glRasterPos2i(20,10); // С позиции экрана (20,10)  
glColor3f(1.0,1.0,0.0); // Желтым цветом  
// Выводим 17 дисплейных списков (17 символов текста)  
glCallLists (17, GL_UNSIGNED_BYTE, "Привет OpenGL !!!");  
...
```

Рассмотрим еще один пример, в котором мы попробуем выбирать любой шрифт из системы Windows и изменять размер выводимых символов текста. Для этого нам придется использовать функцию `GDI CreateFont()`, которая создает логический шрифт с соответствующими атрибутами. Рассмотрим назначение ее параметров:

```
HFONT CreateFont (  
    int nHeight, // Высота фонта (ячейки символа)  
    int nWidth, // Средняя ширина символа  
    int nEscapement, // Угол отношения  
    int nOrientation, // Угол наклона к оси x  
    int fnWeight, // Ширина шрифта  
    DWORD fdwItalic, // Курсив  
    DWORD fdwUnderline, // Подчеркивание  
    DWORD fdwStrikeOut, // Перечеркивание  
    DWORD fdwCharSet, // Идентификатор набора символов  
    DWORD fdwOutputPrecision, // Точность вывода  
    DWORD fdwClipPrecision, // Точность отсечения  
    DWORD fdwQuality, // Качество вывода  
    DWORD fdwPitchAndFamily, // Настройка шага и семейства  
    LPCTSTR lpszFace); // Название шрифта (!)
```



Если значение параметра *nHeight* установить со знаком минус, то мы сообщаем Windows, что надо найти нам шрифт, основанный на высоте *символов*. Если мы используем положительное число, мы выбираем шрифт, основанный на высоте *ячейки*.

Среднюю ширину символа (*nWidth*) обычно задают по умолчанию, для этого достаточно указать в этом поле значение 0.

Параметры *nEscapement* и *nOrientation* задают нулями. Если кому требуется изменять угол наклона символов – экспериментируйте, используя документацию MSDN.

Ширина шрифта (*fnWeight*) может принимать значения из диапазона 0-1000. Но на практике используются только некоторыми значениями, которые и ведены символьные константы: **FW\_DONTCARE** – 0, **FW\_NORMAL** – 400, **FW\_BOLD** – 700, и **FW\_BLACK** – 900. Чем выше значение, тем более жирный шрифт.

Значение TRUE параметра *fdwItalic* разрешает делать курсив.

Значение TRUE параметра *fdwUnderline* разрешает делать подчеркивание.

Значение TRUE параметра *fdwStrikeOut* разрешает делать перечеркивание.

Тип набора символов определяет параметр *fdwCharSet*, например, **RUSSIAN\_CHARSET**, **ANSI\_CHARSET**, **DEFAULT\_CHARSET**, **OEM\_CHARSET**, **SYMBOL\_CHARSET** и т.д.

Параметр *fdwOutputPrecision* сообщает Windows какой из наборов символов использовать, если их доступно больше чем один. Если – **OUT\_TT\_PRECIS**, то система должна выбрать TrueType версию шрифта. TrueType шрифты всегда смотрят лучше, особенно когда масштабируются символы. При использовании константы **OUT\_TT\_ONLY\_PRECIS** система должна использовать только TrueType шрифт.

Точность отсечения (*fdwClipPrecision*) обычно ставится по умолчанию – **CLIP\_DEFAULT\_PRECIS**.

Качество вывода (*fdwQuality*) управляет алгоритмом реалистичности при формировании символов и может принимать значения: **PROOF\_QUALITY**, **DRAFT\_QUALITY**, **NONANTIALIASED\_QUALITY**, **DEFAULT\_QUALITY** или **ANTIALIASED\_QUALITY**. Из контекста названий констант можно определиться с выбором. В нашей программе будем использовать значение **ANTIALIASED\_QUALITY**.

Предпоследний параметр (*fdwPitchAndFamily*) отвечает за шаг и семейство шрифтов. Возможные значения: **FF\_SWISS**, **FF\_DECORATIVE**, **FF\_DONTCARE**, **FF\_MODERN**, **FF\_ROMAN** и **FF\_SCRIPT**.

Для нашего примера можно воспользоваться комбинацией: **FF\_DONTCARE|DEFAULT\_PITCH**.

Последний параметр (*lpszFace*) определяет используемый шрифт.

В данном случае – " Times New Roman Cyr ".

```
...
HFONT hFont; // Windows шрифт
GLuint base; //Базовый номер дисплейного списка
...
void glPrintText(unsigned int base, char *string)
{
    if((base == 0 || string == NULL))
        return;
    glPushAttrib(GL_LIST_BIT);
    glListBase(base - 32);
    glCallLists(strlen(string),GL_UNSIGNED_BYTE, string);
    glPopAttrib();
}
GLvoid CFont(char *fontName, int Size)
{
    base = glGenLists(224); // Выделяем место под 224 символа
    // Создаем шрифт
    hFont = CreateFont(Size, 0, 0, 0, FW_BOLD,
        FALSE, FALSE, FALSE,
        ANSI_CHARSET,
        OUT_TT_PRECIS,
```

```

CLIP_DEFAULT_PRECIS,
ANTIALIASED_QUALITY,
FF_DONTCARE | DEFAULT_PITCH,
fontName);
    SelectObject(hDC, hFont); // Выбираем созданный шрифт
// Форм. 224 дисплейных списков изображений символов, начиная с
// кода 32 и размещая с базового имени дисплейного списка – base
    wglUseFontBitmaps(hDC, 32, 224, base);
    DeleteObject(hFont); // Удаляем шрифт
}
// Инициализация GL
int InitGL(GLvoid) {
    glShadeModel(GL_SMOOTH);
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
    glClearDepth(1.0f);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
// *****Формируем шрифт *****
    CFont("Courier", 36);
    return TRUE; }
GLvoid Draw()
{
    unsigned int ListBase;
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRasterPos2i(20,10);
    glColor3f(1.0,1.0,0.0);
    PrintText(base, "Привет OpenGL!!!");
}

```

Для корректного завершения работы графического приложения рекомендуется удалять из памяти дисплейные списки отведенные под символы:

```

GLvoid KillFont(GLvoid) // Удаление дисплейных списков
{
    glDeleteLists(base, 224); // Удаление 224 символа
}

```

## 10 ПРЕОБРАЗОВАНИЕ КООРДИНАТ

Любое графическое приложение в конечном итоге преобразует координаты вершин графического объекта в оконные координаты графического устройства. В OpenGL это преобразование проходит за несколько основных этапов:

- 1 Модельно-видовое преобразование.
- 2 Преобразование проекции и нормализация.
- 3 Преобразование к области вывода.

### 10.1 Матричные операции

В OpenGL принята правосторонняя система координат, т.е. ось  $z$  направлена на наблюдателя. Для перехода в левостороннюю систему координат необходимы специальные матрицы преобразований.

Все преобразования координат в OpenGL происходят на уровне матричных операций. Все матрицы имеют размер  $4 \times 4$  и участвуют в преобразовании координат по следующему правилу:

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**В OPENGL ВСЕ ОПЕРАЦИИ ПРЕОБРАЗОВАНИЙ ОСУЩЕСТВЛЯЮТСЯ С ТЕКУЩЕЙ МАТРИЦЕЙ. ПРИ ЭТОМ, РАЗЛИЧАЮТ ТРИ ВИДА МАТРИЦ: МОДЕЛЬНО-ВИДОВАЯ, МАТРИЦА ПРОЕКЦИИ И МАТРИЦА ТЕКСТУРЫ. МОДЕЛЬНО-ВИДОВАЯ МАТРИЦА УЧАСТВУЕТ В ПРЕОБРАЗОВАНИЯХ КООРДИНАТ ОБЪЕКТОВ В МИРОВОЙ СИСТЕМЕ КООРДИНАТ, А ИМЕННО, ПЕРЕМЕЩЕНИИ, МАСШТАБИРОВАНИИ И ПОВОРОТАХ. СПОСОБ ПРОЕКЦИИ ГРАФИЧЕСКИХ ОБЪЕКТОВ НА ПЛОСКОСТЬ ЭКРАНА ОПРЕДЕЛЯЕТ МАТРИЦА ПРОЕКЦИОНОВАНИЯ. СВЯЗЬ КООРДИНАТ ТОЧЕК ИЗОБРАЖЕНИЯ ТЕКСТУРЫ С КООРДИНАТАМИ ТОЧЕК ПОВЕРХНОСТИ ГРАФИЧЕСКОГО ОБЪЕКТА ОПРЕДЕЛЯЕТ МАТРИЦА ТЕКСТУР.**

**ОПРЕДЕЛЕНИЕ ТЕКУЩЕЙ МАТРИЦЫ ПРЕОБРАЗОВАНИЙ ОСУЩЕСТВЛЯЕТСЯ КОМАНДОЙ: VOID GLMATRIXMODE (GLenum MODE), ГДЕ MODE ПРИНИМАЕТ ЗНАЧЕНИЕ КОНСТАНТ СОГЛАСНО ТАБЛИЦЫ.**

<i>Mode</i>	Вид матрицы
<b>GL_MODELVIEW</b>	Модельно-видовая
<b>GL_PROJECTION</b>	Проекция
<b>GL_TEXTURE</b>	Текстуры

**Внимание!** Умножение координат объекта на текущую матрицу происходит в момент вызова соответствующей команды OpenGL, определяющей координату.

Для заполнения текущей матрицы преобразований соответствующими значениями предназначена специальная команда:

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

Например, для формирования единичной матрицы можно воспользоваться следующим кодом

```
GLfloat mId[4][4] = { 1.0f, 0.f, 0.0f, 0.0f,
                    0.0f, 1.0f, 0.0f, 0.0f,
                    0.0f, 0.0f, 1.0f, 0.0f,
                    0.0f, 0.0f, 0.0f, 1.0f};
```

```
glLoadMatrixf(&mId[0][0]);
```

**Внимание!** В массив матрица записывается *по столбцам*.

Кстати, такой вид записи матрицы в программе хорошо согласуется с общепринятым подходом в компьютерной графике, когда точка в пространстве задается вектором строкой. В этом случае при любом преобразовании вектор-строка умножается на матрицу преобразования, что требует изменения расстановки элементов в матрице (строки меняются местами со столбцами). Например, для задачи одновременного увеличения масштаба геометрического объекта по оси *y* в 2 раза и смещении его точек по оси *x* на 3 единицы, по оси *z* на 4 единицы достаточно сформировать массив в виде

```
GLfloat mST[4][4] = { 1.0f, 0.f, 0.0f, 0.0f,
                    0.0f, 2.0f, 0.0f, 0.0f,
                    0.0f, 0.0f, 1.0f, 0.0f,
                    3.0f, 0.0f, 4.0f, 1.0f};
```

и передать его в текущую матрицу преобразования

```
glLoadMatrixf(&mST[0][0]);
```

**Внимание!** Примите этот дополнительный пример, как рекомендацию к использованию на практике для написания высокоэффективного кода графической программы.

Единичная матрица часто используется в OpenGL и для облегчения программирования предусмотрена команда `glLoadIdentity()`, которая устанавливает единичную матрицу текущего преобразования.

Иногда приходится умножить текущую матрицу на другую матрицу. Для этой цели используется команда `void glMultMatrix[f d] (GLtype *m)`, где параметр *m* задает матрицу размером 4×4, на которую будет умножаться текущая матрица.

## СОСТОЯНИЕ МАТРИЦ ПРЕОБРАЗОВАНИЯ МОЖНО СОХРАНЯТЬ И ВЫЗЫВАТЬ ПО МЕРЕ НАДОБНОСТИ. ДЛЯ ЭТО СУЩЕСТВУЮТ КОМАНДЫ:

`void glPushMatrix (void)`

`void glPopMatrix (void)`

Эти команды сохраняют и читают текущую матрицу преобразования из стека. Для каждого вида матриц преобразований имеется свой стек.

### 10.2 Модельно-видовые преобразования

**ОПРЕДЕЛЕНИЕ МАТРИЦЫ ВИДОВОГО ПРЕОБРАЗОВАНИЯ УДОБНЕЕ ОСУЩЕСТВЛЯТЬ СПЕЦИАЛЬНЫМИ КОМАНДАМИ OPENGL: ПЕРЕНОС, МАСШТАБИРОВАНИЕ И ВРАЩЕНИЕ. ПРЕОБРАЗОВАНИЯ ОСУЩЕСТВЛЯЮТСЯ В МИРОВОЙ СИСТЕМЕ КООРДИНАТ. РАССМОТРИМ ИХ ПОДРОБНЕЕ.**

`void glTranslate[f d] (GLtype Dx, GLtype Dy, GLtype Dz)`

**ЭТА КОМАНДА ГОТОВИТ МАТРИЦУ ПРЕОБРАЗОВАНИЯ ДЛЯ ПЕРЕНОСА ГРАФИЧЕСКОГО ОБЪЕКТА НА РАССТОЯНИЕ  $Dx$  ПО ОСИ  $X$ , НА РАССТОЯНИЕ  $Dy$  ПО ОСИ  $Y$  И НА РАССТОЯНИЕ  $Dz$  ПО ОСИ  $Z$ .**

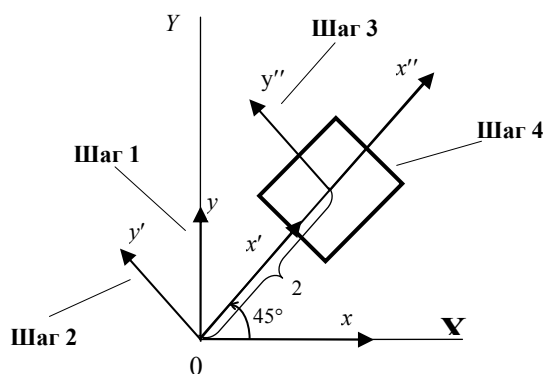
`void glScale[f d] (GLtype Sx, GLtype Sy, GLtype Sz)`

**ПРИ ПОМОЩИ ЭТОЙ КОМАНДЫ ОСУЩЕСТВЛЯЕТСЯ ФОРМИРОВАНИЕ ВИДОВОЙ МАТРИЦЫ ДЛЯ МАСШТАБИРОВАНИЯ ОБЪЕКТА ВДОЛЬ ОСЕЙ МИРОВЫХ КООРДИНАТ. В  $Sx$  РАЗ ВДОЛЬ ОСИ  $X$ , В  $Sy$  РАЗ ВДОЛЬ ОСИ  $Y$  И В  $Sz$  РАЗ ВДОЛЬ ОСИ  $Z$ .**

**VOID GLROTATE[F D](GLTYPE ANGLE, GLTYPE X, GLTYPE Y, GLTYPE Z)**

Эта команда рассчитывает видовую матрицу для выполнения вращения объекта против часовой стрелки на угол  $angle$  относительно радиус-вектора, заданного параметрами  $x$ ,  $y$  и  $z$ . **Внимание!** Угол  $angle$  задается в градусах.

*Необходимо помнить*, что на самом деле преобразования осуществляются не над самим объектом, а над его локальной системой координат. Это очень удобно, когда в одну глобальную систему координат сцены необходимо вставить определенным образом несколько объектов, описанных в своих системах локальных координат. Поясним на примере (все рассматриваем в плоскости  $z = 0$ ). Пусть требуется объект **A** (квадрат), заданный в своей системе координат, переместить вдоль оси  $x$  на 2 единицы и развернуть на  $45^\circ$  против часовой стрелки относительно оси  $z$  (см. рис. 2). Такая постановка задачи в OpenGL не применима, так как объект определяется только в конце своих геометрических преобразований. Правильнее было сказать: необходимо развернуть на  $45^\circ$  (против часовой стрелки) систему координат относительно вектора  $(0,0,1)$ , после чего, переместить ее (относительно старого положения) в новое место вдоль вектора  $(2,0,0)$  и определить объект **A** в новом положении системы координат.



**Рис. 2** Пояснение модельно-видовых преобразований в OpenGL

Для выполнения такой задачи можно использовать следующий фрагмент программы модельно-видового преобразования:

```

//шаг 1 – устанавливается единичная текущая матрица (оси x0y)
glLoadIdentity();
//шаг 2 – поворачивается система координат x0y на 45°
// против часовой стрелки относительно вектора
//(0,0,1) и занимает новое положение x'0y'
glRotatef(45,0,0,1);
//шаг 3 – система x'0y' смещается по оси x' на 2 единицы и
// занимает положение x''0y''
glTranslatef(2.0f, 0.0f, 0.0f);
//шаг 4 – определяем объект и текущая матрица
// модельно-видового преобразования умножается
// на координаты объекта
glBegin(GL_QUADS);
glVertex3f(-1.0f, 1.0f, 0.0f); // левый верхний угол
glVertex3f( 1.0f, 1.0f, 0.0f); // правый верхний угол
glVertex3f( 1.0f, -1.0f, 0.0f); // правый нижний угол
glVertex3f(-1.0f, -1.0f, 0.0f); // левый нижний угол
glEnd();

```

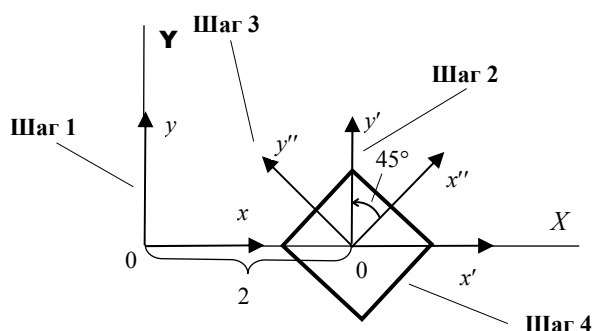
Теперь рассмотрим, что произойдет с объектом **A**, если поменять местами шаг 2 с шагом 3 в приведенном выше фрагменте программы? Получим новый код.

```

// Устанавливается единичная текущая матрица оси x0y
glLoadIdentity();
// Система x0y смещается по оси x на 2 единицы и занимает
// положение x'0y'
glTranslatef(2.0f, 0.0f, 0.0f);
// Поворачивается система координат x'0y' на 45°
// против часовой стрелки относительно вектора (0,0,1) и
// занимает новое положение x''0y''
glRotatef(45,0,0,1);
// Определяем объект и текущая матрица модельно-видового
// преобразования умножается на координаты объекта
glBegin(GL_QUADS);
glVertex3f(-1.0f, 1.0f, 0.0f); // левый верхний угол
glVertex3f( 1.0f, 1.0f, 0.0f); // правый верхний угол
glVertex3f( 1.0f, -1.0f, 0.0f); // правый нижний угол
glVertex3f(-1.0f, -1.0f, 0.0f); // левый нижний угол
glEnd();

```

Конечный результат такого преобразования будет отличаться предыдущей задачи. Ход преобразований можно проанализировать на рисунке 3.



**Рис. 3** Пример модельно-видовых преобразований

Если в приведенном примере (рис. 4) рассуждать по геометрическим преобразованиям с точки зре-

ния объекта, то конечную цель можно достичь после следующих шагов: шаг 1 – поворота объекта на 45° относительно оси  $z$  после чего; шаг 2 – перемещение объекта (повернутого) вдоль оси  $x$  на две единицы. Но в OpenGL это осуществляется в обратном порядке.

*Совет:* приучите себя к рассуждениям о преобразованиях объектов с точки зрения их систем координат, и это вас избавит от возможных геометрических ошибок.

Напоминаем, что текущая матрица умножается на координаты точек объекта только при определении самого объекта. После чего координаты объекта преобразуются в новые значения. А до этого момента формируется текущая матрица преобразования. Любой вызов команды преобразования завершается перемножением текущей матрицы преобразования на сформированную матрицу командой преобразования. Полученная матрица размещается на место текущей.

Изменяя положение самого объекта, создается иллюзия, что перемещается камера наблюдения. В OpenGL существует возможность управлять камерой с одновременным изменением модельно-видовой матрицы. Это можно сделать с помощью команды:

```
void gluLookAt( GLdouble eyex, GLdouble eyez, GLdouble eyez,
               GLdouble cx, GLdouble cy, GLdouble cz,
               GLdouble upx, GLdouble upy, GLdouble upz)
```

**ЗДЕСЬ ПАРАМЕТРЫ  $EYEX$ ,  $EYEY$ ,  $EYEZ$  ОПРЕДЕЛЯЮТ ТОЧКУ НАБЛЮДЕНИЯ,  $CX$ ,  $CY$ ,  $CZ$  ЗАДАЮТ ЦЕНТР СЦЕНЫ, КОТОРЫЙ БУДЕТ ПРОЕКТИРОВАТЬСЯ В ЦЕНТР ОБЛАСТИ ВЫВОДА, И ПАРАМЕТРЫ  $UPX$ ,  $UPY$ ,  $UPZ$  ЗАДАЮТ ВЕКТОР ПОЛОЖИТЕЛЬНОГО НАПРАВЛЕНИЯ ОСИ  $U$  СЦЕНЫ, ОПРЕДЕЛЯЯ ПОВОРОТ КАМЕРЫ. ОБЫЧНО ПАРАМЕТРЫ  $UPX$ ,  $UPY$ ,  $UPZ$  ИМЕЮТ ЗНАЧЕНИЯ – (0,1,0).**

**Внимание!** Команду gluLookAt() обычно обрабатывают, когда модельно-видовая матрица равна единичной. Например:

```
...
glMatrixMode(GL_MODELVIEW); // текущая матрица – видовая
glLoadIdentity(); // текущая матрица – единичная
gluLookAt(2.0f, 1.0f, 3.0f, // положение центра наблюдения
          6.0f, 0.0f, 0.0f, // центр сцены в мировых координатах
          0.0f, 1.0f, 0.0f); // вектор "верх" направлен вдоль оси y
...

```

### 10.3 Область вывода

В OpenGL готовая для визуализации графическая информация поступает в специально настроенную область вывода, которая определяется шириной ( $p_x$ ), высотой ( $p_y$ ) и положением ее центра на экране ( $o_x$ ,  $o_y$ ). Эти параметры формируются специальной командой:

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

Параметры  $x$  и  $y$  задают координаты левого нижнего угла ( $x$ ,  $y$ ) области вывода в оконной системе координат. Параметры  $width$  и  $height$  определяют ширину и высоту окна вывода. Все параметры команды задаются в пикселях. Размеры оконной системы координат определяются текущими размерами окна приложения, точка (0,0) находится в левом нижнем углу окна.

После выполнения команды glViewport параметры области вывода устанавливаются в значения  $p_x = width$ ,  $p_y = height$ ,  $o_x = x + width/2$  и  $o_y = y + height$  (точка с координатами (0, 0, 0) в оконных координатах будет располагаться в центре окна области вывода), а оконные координаты выводимых вершин определяются соотношением:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} x \frac{p_x}{2} + o_x \\ y \frac{p_y}{2} + o_y \\ z \frac{f-n}{2} + \frac{n+f}{2} \end{pmatrix},$$

где  $n$  и  $f$  определяют допустимый диапазон изменения глубины ( $z$  – координата). По умолчанию данные параметры равны 0 и 1 соответственно. Но их можно изменять командой:

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

После отсечения и деления на  $w$  координаты  $z$  лежат в диапазоне значений от  $-1$  до  $1$ , соответственно для ближней и дальней плоскостей отсечения. Команда `glDepthRange` определяет линейное отображение нормализованных  $z$ -координат из этого диапазона в оконные координаты.

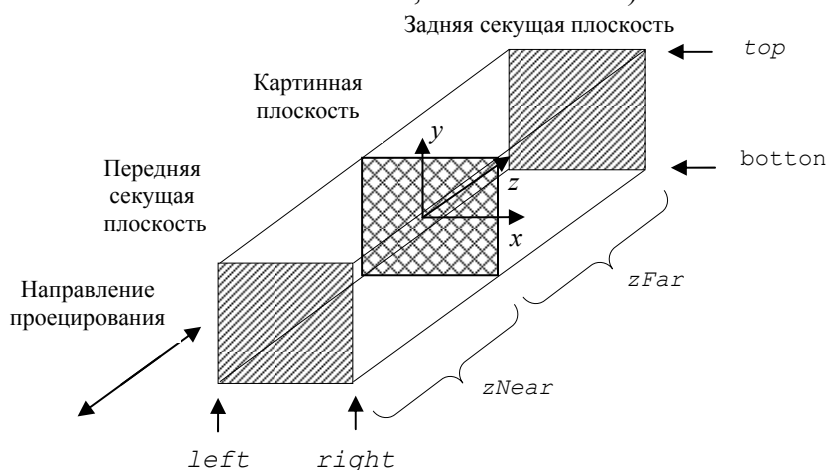
Команду `glViewport` часто используют в CAD системах для моделирования четырех видов проектируемого объекта. Это можно реализовать, например, следующим образом: экран разбивается на четыре части и каждая описывается командой `glViewport` с соответствующими значениями параметров окна вывода. После каждого такого описания необходимо настроить вид проекции и параметры камеры наблюдения. После чего, необходимо вывести сцену. И так четыре раза.

## 11 ПРОЕКЦИИ

В OpenGL командно реализованы общепринятые в компьютерной графике два вида проекций: ортографическая (параллельная) и перспективная с одной главной точкой схода на оси  $z$ . Для реализации других видов проекций необходимо использовать готовые матрицы проецирования.

### 11.1 Ортографическая проекция

Первый тип проекции (рис. 4) может быть задан командой  
`void glOrtho (GLdouble left, GLdouble right,`  
`GLdouble bottom, GLdouble top,`  
`GLdouble zNear, GLdouble zFar)`



**Рис. 4 Усеченный объем видимости для случая ортографической параллельной проекции**

Команда `glOrtho()` создает матрицу параллельной проекции и усеченный объем видимости. Параметры `left` и `right` задают координаты левой и правой вертикальной плоскости отсечения. Параметры `bottom` и `top` определяют координаты верхней и нижней плоскости отсечения. Параметры `zNear` и `zFar` задают расстояние от точки  $(0,0,0)$  до ближней и дальней плоскостей отсечения и могут быть положительными и отрицательными.

В процессе работы команды `glOrtho()` текущая матрица  $M$  умножается на матрицу, сформированную командой `glOrtho()`, и результат помещается на место матрицы  $M$ .

В библиотеке `glu32.lib` существует эквивалент команды `glOrtho(left,right,bottom,top, -1, 1)`:

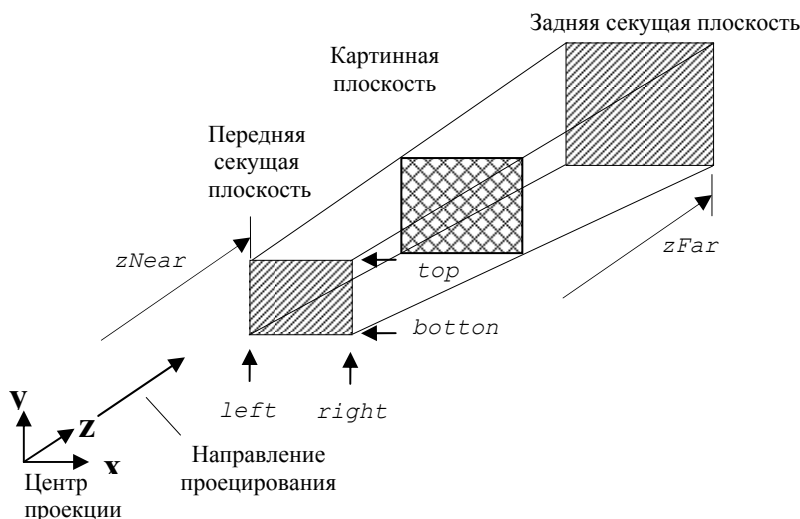
```
void gluOrtho2D (GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top)
```

### 11.2 Перспективная проекция

Для задания перспективной проекции с одной главной точкой схода по оси  $z$  применяется команда

```
void glFrustum (GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top,
GLdouble zNear, GLdouble zFar)
```

Команда `glFrustum()` создает матрицу центральной проекции и усеченный пирамидальный объем видимости (рис. 5). Центр проецирования



**Рис. 5 Перспективная проекция и усеченный пирамидальный объем видимости**

(точка наблюдения) находится в начале системы координат. Параметры *left* и *right* задают координаты левой и правой вертикальной плоскости отсечения. Параметры *bottom* и *top* определяют координаты верхней и нижней плоскости отсечения. Параметры *zNear* и *zFar* задают расстояние от наблюдателя до ближней и дальней плоскостей отсечения, причем оба значения должны быть положительными. Параметры *left*, *right*, *bottom* и *top* задаются для ближней плоскости отсечения.

**Внимание!** В конечном итоге значение глубины (*z*) любой выводимой точки сцены нормируется к возможностям буфера глубины, т.е. к значению в диапазоне от 0 до 1. Чем больше значение отношения  $zNear/zFar$ , тем менее эффективно будут различаться в буфере глубины, расположенные рядом поверхности. Не следует задавать параметру *zNear* нулевое значение, так как в этом случае отношение  $zNear/zFar$  стремиться к бесконечности.

В библиотеке `glu32.lib` существует команда `gluPerspective()` аналогичная по функциям, что и `glFrustum()`, но с различными входными параметрами.

```
void gluPerspective (GLdouble fovy, GLdouble aspect,
                    GLdouble zNear, GLdouble zFar)
```

Данная команда задает усеченный пирамидальный объем видимости в видовой системе координат. Параметр *fovy* определяет угол видимости (в градусах) вдоль оси *y*. Угол видимости вдоль оси *x* задается параметром *aspect*, который определяется как отношение ширины и высоты области вывода. Параметры *zNear* и *zFar* задают расстояние от наблюдателя до ближней и дальней плоскостей отсечения, причем оба значения должны быть положительными. Эквивалентность работы данной команды с командой `glFrustum()`, наступает при

$$left = -right, bottom = -top, tg(fovy/zNear) = top/zNear, aspect = right/top.$$

Для того чтобы включить проекцию необходимо включить режим проецирования и загрузить единичную текущую матрицу преобразования. Например:

```
...
// Установка настроек области вывода
glViewport(0,0,width,height); // Выбираем матрицу проекции
glMatrixMode(GL_PROJECTION); // Делаем ее единичной
glLoadIdentity();
gluPerspective(45.0f,width/height,0.1f,100.0f);
```



```
glMatrixMode(GL_MODELVIEW); // Делаем ее единичной
glLoadIdentity();
```

...

## 12 ОБРАТНЫЕ ПРЕОБРАЗОВАНИЯ ОКОННЫХ КООРДИНАТ В ОБЪЕКТНЫЕ

Часто требуется пользователю, используя мышь или другой указатель, выбрать точку в трехмерном пространстве, оперируя лишь ее проекцией на экране. Решая данную задачу, необходимо вспомнить, что оконные координаты объекта получаются из его мировых путем трех матричных преобразований: модельно-видовых, проекционных и преобразований к области вывода. И для определения мировых координат точки объекта по его оконным координатам требуется проделать обратные, вышеприведенные, преобразования. Но тут возникает другая проблема, множество точек из мировых координат, расположенных вдоль луча-проектора, дадут одну и ту же проекцию на экране. Для разрешения этой неоднозначности в OpenGL обратные преобразования решены командами **gluUnProject()** при диапазоне глубин в вашем приложении [0, 1], либо **gluUnProject4()**, если диапазон глубин отличается от [0, 1]. Рассмотрим параметры и функциональные возможности этих команд:

```
int gluUnProject (GLdouble winx, GLdouble winy, GLdouble winz,
const GLdouble modelMatrix[16],const GLdouble projMatrix[16],
const GLint viewport[4],
GLdouble *objx, GLdouble *objy, GLdouble *objz);
```

Данная команда преобразует заданные оконные координаты (*winx*, *winy*, *winz*) в объектные координаты (*objx*, *objy*, *objz*), используя преобразования, заданные модельно-видовой матрицей (*modelMatrix*), проекционной матрицей (*projMatrix*) и портом просмотра (*viewport*). Если функция выполнилась удачно, то она возвращает значение `GL_TRUE`, в противном случае `GL_FALSE`.

Для получения однозначного результата от обратного преобразования, команда **gluUnProject()** требует в качестве одного из своих параметров значение глубины в оконных координатах (*winz*). Если *winz* = 0.0, то вызов **gluUnProject()** вернет координаты точки (*objx*, *objy* и *objz*), расположенной на ближней плоскости отсечения объема видимости, а при *winz* = 1.0 будет вычислена точка на дальней плоскости отсечения.

Изъятие текущих значений матриц преобразований осуществляется командой:

```
void glGetDoublev( GLenum pname, GLdouble *params)
```

Параметр *pname* определяет вид матрицы (см. документацию):

модельно-видовой – `GL_MODELVIEW_MATRIX`;

проецирования – `GL_PROJECTION_MATRIX`. Параметр *params* указывает на массив размещения.

Значения параметров окна можно выбрать командой:

```
void glGetIntegerv( GLenum pname, GLdouble *params)
```

Назначение параметров такое же, как и в команде **glGetDoublev**. Значение параметра *pname* для окна должно быть `GL_VIEWPORT`.

Рассмотрим фрагмент программы, где по экранным координатам точки(*x* и *y*), например, взятых от мыши, вычисляем значение координат точки объекта, обеспечивших эту проекцию. В качестве *winz* возьмем значение 0.0.

...

```
GLint viewport[4]; //для окна
```

```
GLdouble mv_matrix[16]; // для модельно-видовой матрицы
```

```
GLdouble proj_matrix[16]; // для матрицы прецирования
```

```
GLdouble objx,objy,objz; // Возвращаемые объектные x, y, z координаты
```

```
GLint x,y;
```

...

```
// Обрабатываем событие от "мышки" (формируем значение x и y)
```

...

```
glGetIntegerv(GL_VIEWPORT,viewport);
```

```
glGetDoublev(GL_MODELVIEW_MATRIX,mv_matrix);
```

```
glGetDoublev(GL_PROJECTION_MATRIX,proj_matrix);
gluUnProject(x,y,0.0,mv_matrix,proj_matrix,viewport,&ox,&oy,&oz);
printf("Объектные координаты при z=0 (%f,%f,%f)\n",ox,oy,oz);
```

...

Подобные ситуации часто возникают при выборе объектов на экране.

### 13 ПОСТРОЕНИЕ РЕАЛИСТИЧЕСКИХ ИЗОБРАЖЕНИЙ

Компьютерное графическое моделирование реальных либо виртуальных объектов предполагает воссоздание на экране компьютера не только правильные геометрические соотношения элементов сцены, но и передачу "атмосферы" визуального ряда. Сюда включаются: внешний вид материала, из которого сделан объект, признаки наличия света; преломление и смешивание цветов через полупрозрачные среды фрагментов объекта, изменяющийся по глубине туман и т.д. Для создания реалистических изображений в OpenGL существует богатый командный инструментарий, обеспечивающий:

- наложение текстур на поверхность объекта,
- задание свойств материала из которого сделан объект,
- определение свойств источников света,
- задание модели освещения,
- задание законов смешивания цветов через полупрозрачные поверхности,
- задание тумана (дымки) и т.д.

#### 13.1 Текстуры

Текстурой называется одномерное или двумерное изображение с совокупностью параметров, определяющих, каким образом производится наложение изображения на поверхность графического объекта.

В OpenGL текстурирование выполняется по следующей схеме:

1 Размещаем в оперативной памяти массив данных, в котором будет храниться образ будущей текстуры.

2 Выбираем изображение (например, из файла) и преобразуем его к внутреннему формату OpenGL. Заполняем массив данными изображения.

3 Создаем текстуру в памяти. Для этого будем использовать команду `glTexImage2D()` для двумерных текстур, либо `gluBuild2Mipmaps()`.

4 Задаем возможные параметры фильтрации текстуры для достижения наилучшего восприятия объектов при различных коэффициентах масштабирования. Для этого будем использовать команду `glTexParameter()`.

5 Указываем каким образом координаты текстуры связанные с координатами текстурируемого объекта. Это можно выполнить, например, командой `glTexCoord()`.

Рассмотрим этапы текстурирования объекта подробнее.

##### 13.1.1 Подготовка текстуры

В связи с тем, что OpenGL работает с многими операционными системами, то разработчики графической библиотеки ввели свой формат для хранения изображения. В этом формате любое изображение представлено в виде последовательности RGB составляющих точек изображения. Для облегчения считывания графических данных из файла, преобразования их во внутренний формат OpenGL и размещения в памяти существует функция `AUX_RGBImageRec *auxDIBImageLoad(const char *Filename)`, где *Filename* – название файла с расширением \*.bmp или \*.dib. Функция возвращает указатель на область памяти, где хранятся преобразованные данные файла. Для работы с данной функцией необходимо подключить библиотеку `glaux.lib`

Рассмотрим пример загрузки изображения будущей текстуры:

...

```
AUX_RGBImageRec *texture;
texture = auxDIBImageLoad("Image/t1.bmp");
```

...

В данном примере происходит загрузка файла картинки "t1.bmp" из каталога "Image". Данные файла будут сохранены в структуре `texture`, которую мы задали с помощью `AUX_RGBImageRec`.

Следует учитывать (!), что для корректной работы с текстурами необходимо соблюдать условие:

размеры изображения должны быть равны степени двойки. Например: 64×64, 128×64, 256×128 и т.п. Для обеспечения этого условия можно предварительно обработать изображение в среде графического редактора, подобрав размеры "картинки" под стандарт OpenGL, либо программно изменить масштабы изображения.

Для программного изменения масштабом изображения существует команда

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

Рассмотрим параметры команды:

Параметр	Описание
<i>format</i>	Определяет формат данных пикселя и может принимать значения: <b>GL_RED</b> , <b>GL_GREEN</b> , <b>GL_BLUE</b> , <b>GL_RGB</b> , <b>GL_ALPHA</b> , <b>GL_RGBA</b> , <b>GL_COLOR_INDEX</b> и т.д. (см. док.). На практике обычно используют значения <b>GL_RGB</b> или <b>GL_RGBA</b>
<i>widthin</i>	Задаёт ширину оригинала изображения
<i>heightin</i>	Задаёт высоту оригинала изображения
<i>typein</i>	Задаёт тип данных для оригинала изображения, например, <b>GL_UNSIGNED_BYTE</b> , <b>GL_BYTE</b> , <b>GL_BITMAP</b> , <b>GL_UNSIGNED_SHORT</b> , <b>GL_INT</b> , <b>GL_UNSIGNED_INT</b> , <b>GL_SHORT</b> , или <b>GL_FLOAT</b>

Продолжение табл.

Параметр	Описание
<i>datain</i>	Указатель на данные оригинала изображения
<i>widthout</i>	Задаёт ширину измененного масштабом изображения
<i>heightout</i>	Задаёт высоту измененного масштабом изображения
<i>typeout</i>	Задаёт тип данных для измененного масштабом изображения (см. <i>typein</i> )
<i>dataout</i>	Указатель на данные измененного масштабом изображения

В результате своей работы функция `gluScaleImage()` берет из памяти по адресу *datain* изображение "картинки", изменяет его, согласно своих параметров, и заносит в область памяти, на которую указывает параметр *dataout*. В случае удачного преобразования функция возвращает значение ноль, либо код ошибки.

### 13.1.2 Создание текстуры

После того как изображение для двумерной текстуры подготовлено и находится в памяти, можно создать саму текстуру. Это осуществляется командой:

```
void glTexImage2D (GLenum target, GLint level, GLint interformat,
                 GLsizei width, GLsizei height,
                 GLint border, GLenum format,
                 GLenum type, const GLvoid *pixels)
```

Данная команда работает только в режиме RGBA. Рассмотрим ее параметры:

Параметр	Описание
<i>target</i>	Определяет тип создаваемой текстуры и должен быть равен <b>GL_TEXTURE_2D</b> .
<i>level</i>	Задаёт число уровней детализации текстуры. Для базового уровня – 0, а если уменьшенный в <i>n</i> раз, то <i>n</i> .
<i>interformat</i>	Задаёт число цветных компонентов текстуры и может принимать значения: 1 – если используется только красный (R) компонент цвета; 2 – если красный (R) и зелёный (G); 4 – если красный (R), зелёный (G) и синий (B); 4 – если все компоненты (RGBA).

Продолжение табл.

Параметр	Описание
<i>width</i>	Определяет ширину изображения текстуры и должен вычисляться по формуле $2^n + 2 \times border$
<i>height</i>	Определяет высоту изображения текстуры и должен вычисляться по формуле $2^m + 2 \times border$
<i>border</i>	Задаёт ширину границы изображения текстуры. Принимаемые значения: 0 или 1.
<i>format</i>	Определяет формат данных пикселя и может принимать одно из значения: <b>GL_RED</b> , <b>GL_GREEN</b> , <b>GL_BLUE</b> , <b>GL_ALPHA</b> , <b>GL_RGB</b> , <b>GL_RGBA</b> , <b>GL_BGR_EXT</b> , <b>GL_COLOR_INDEX</b> , <b>GL_LUMINANCE_ALPHA</b> , <b>GL_BGRA_EXT</b> , и <b>GL_LUMINANCE</b> . Последние два значения преобразуют изображение текстуры в монохромное, так как в этом случае любая составляющая цвета переводится в яркостную.
<i>type</i>	Задаёт тип данных пикселей (см. <code>gluScaleImage()</code> )
<i>pixels</i>	Определяет указатель на данные изображения для текстуры

Приведем пример вызова команды `glTexImage2D()`:

```
glTexImage2D(GL_TEXTURE_2D, 0, 3,
             Texture->sizeX, Texture->sizeY,
             0, GL_RGB, GL_UNSIGNED_BYTE,
             Texture->data);
```

В данном случае создается двумерная текстура (**GL\_TEXTURE\_2D**). С нулевым уровнем детализации (0). Изображение сделано из трех компонент R, G и B (3). Ширина текстуры определяется как `Texture->sizeX`. `Texture->sizeY` – высота текстуры. Ширина бордюра изображения равна нулю (0). Компоненты цвета располагаются в последовательности R, G и B (**GL\_RGB**). Изображение состоит из байт без знака (**GL\_UNSIGNED\_BYTE**). `Texture->data` указывает команду, где брать данные изображения.

Как известно, для качественного текстурирования объектов необходимо соблюдать масштабные пропорции между текстурой и покрываемым объектом. Чем больше увеличение объекта, тем больше требуется образ текстуры для покрытия. И, наоборот, чем меньше становится объект тем меньше требуется размер текстуры. Вот тут и требуется уровень детализации текстуры (mipmap). Если текстура имеет размер  $2^n \times 2^m$ , то можно построить  $\max(n, m) + 1$  уменьшенных образов текстур. Команда `glTexImage2D()` позволяет определить  $\max(n, m)$  уменьшенных текстур, в каждой из которой будет храниться уменьшенная копия оригинала. Благодаря этому OpenGL будет использовать соответствующий уровень детализации текстуры при соответствующем масштабе объекта.

В OpenGL имеется специальная команда (для одномерных текстур не рассматриваем) успешно выполняющая автоматическое построение всех возможных уровней уменьшающихся текстур по отношению к оригиналу. Это команда `gluBuild2DMipmaps()`:

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

Назначение параметров соответствует команде `glTexImage2D()`. Кроме основной функции эта команда может работать с изображениями текстур размер которых не является степенью 2. В этом случае происходит масштабирование образа до ближайших степеней 2. При успешном завершении работы команда возвращает значение 0, а в противном случае – код ошибки.

После выполнения команд создания текстур, последние копируются во внутреннюю память OpenGL, и поэтому память, занимаемую исходным изображением, можно освободить.

Перед тем как приступить к созданию текстур необходимо разрешить соответствующий режим командой `glEnable(GL_TEXTURE_2D)`.

### 13.1.3 Эстетические параметры текстурирования объектов

При текстурировании поверхностей 3D моделей возникают множество проблем с качеством визуализации текстур. В основном проблемы возникают когда на одну точку изображения на экране приходится несколько точек текстуры и наоборот. Это происходит при приближении или удалении от камеры наблюдаемого объекта. А бывают ситуации, когда в центральной проекции поверхность 3D объекта находится своими участками одновременно и в зоне увеличения и уменьшения масштаба объекта, например, этот эффект хорошо наблюдается когда камера, расположенная рядом с поверхностью, смотрит вдоль ее. В любом случае, если не предпринимать специальных мер, наблюдается лестничный эффект для близких поверхностей и потеря рисунка текстуры для дальних участков поверхности.

Для устранения подобных проявлений в визуализации текстур в OpenGL имеются команды настраивающие параметры визуализации текстур:

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

ГДЕ:

**TARGET** ОПРЕДЕЛЯЕТ ТИП ТЕКСТУРЫ И ПРИНИМАЕТ ЗНАЧЕНИЯ: `GL_TEXTURE_2D` ИЛИ `GL_TEXTURE_1D`;

**PNAME** ОПРЕДЕЛЯЕТ НАЗВАНИЕ ПАРАМЕТРА ТЕКСТУРЫ, КОТОРОЕ БУДЕТ ИЗМЕНЯТЬСЯ;

**PARAM (PARAMS)** УСТАНОВЛИВАЕТ НОВОЕ ЗНАЧЕНИЕ ПАРАМЕТРА **PNAME**;  
РАССМОТРИМ ВОЗМОЖНЫЕ ПАРАМЕТРЫ ТЕКСТУРЫ (**PNAME**):

<i>pname</i>	Описание
<code>GL_TEXTURE_MIN_FILTER</code>	• ОПРЕДЕЛЯЕТ АЛГОРИТМ СЖАТИЯ ТЕКСТУРЫ ДЛЯ ОБЪЕКТОВ, РАЗМЕР КОТОРЫХ

<i>pname</i>	Описание
	<b>БЫЛ УМЕНЬШЕН. СУЩЕСТВУЕТ ШЕСТЬ ТАКИХ АЛГОРИТМОВ. ПРИ ЗНАЧЕНИИ GL_NEAREST БУДЕТ ИСПОЛЬЗОВАТЬСЯ ОДНА БЛИЖАЙШАЯ ТОЧКА В ТЕКСТУРЕ, А ПРИ ЗНАЧЕНИИ GL_LINEAR ЧЕТЫРЕ БЛИЖАЙШИХ ТОЧЕК ТЕКСТУРЫ. ОСТАЛЬНЫЕ ЧЕТЫРЕ ОПРЕДЕЛЯЮТ УРОВНИ ДЕТАЛИЗАЦИИ ДЛЯ MIPMAPPING. ПО УМОЛЧАНИЮ ЗНАЧЕНИЕ ФУНКЦИИ – GL_LINEAR.</b>
<b>GL_TEXTURE_MAG_FILTER</b>	• Определяет функцию увеличения текстуры для текстурирования объектов, размер которых был увеличен или нет. Существует два таких алгоритма. При значении <b>GL_NEAREST</b> будет использоваться одна ближайшая точка, а при значении <b>GL_LINEAR</b> четыре ближайших элемента текстуры. По умолчанию значение функции – <b>GL_LINEAR</b> .
<b>GL_TEXTURE_WRAP_S</b>	• Определяет сворачивание координаты s в текстуре. При значении <b>GL_CLAMP</b> используются фиксированные значения: 0 или 1, что необходимо в случае когда накладывается один образ текстуры на объект. При значении <b>GL_REPEAT</b> координата s может задаваться любым значением и изображение текстуры размножается вдоль координаты s по поверхности объекта. По умолчанию значение функции – <b>GL_REPEAT</b> .
<b>GL_TEXTURE_WRAP_T</b>	• Определяет координаты t в текстуре аналогично <b>GL_TEXTURE_WRAP_S</b> .

**ПРИ ИЗМЕНЕНИИ РАЗМЕРОВ ОБЪЕКТОВ ПО ОТНОШЕНИЮ К ТЕКСТУРЕ СУЩЕСТВУЮТ СЛЕДУЮЩИЕ ШЕСТЬ АЛГОРИТМОВ "ПОДГОНКИ" ТЕКСТУРЫ К НОВЫМ РАЗМЕРАМ ОБЪЕКТА:**

<i>param (params)</i>	Описание
<b>GL_NEAREST</b>	<b>ВЫБИРАЕТ БЛИЖАЙШУЮ ТОЧКУ В ТЕКСТУРЕ</b>

<i>param</i> ( <i>params</i> )	Описание
<b>GL_LINEAR</b>	Возвращает среднеарифметическое значение четырех ближайших точек в текстуре
<b>GL_NEAREST_MIPMAP_NEAREST</b>	Выбирает уровень текстурной детализации наиболее близко соответствующий размеру пикселя. При этом выбор точек текстуры осуществляется по алгоритму <b>GL_NEAREST</b>
<b>GL_LINEAR_MIPMAP_NEAREST</b>	Выбирает уровень текстурной детализации наиболее близко соответствующий размеру пикселя. При этом выбор точек текстуры осуществляется по алгоритму <b>GL_LINEAR</b>
<b>GL_NEAREST_MIPMAP_LINEAR</b>	Выбирает два уровня текстурной детализации наиболее близко соответствующих размеру пикселя. При этом выбор точек текстуры в каждом текстурном слое осуществляется по алгоритму <b>GL_NEAREST</b> . Итоговым значением текстуры является среднеарифметическое значение двух выбранных точек текстуры
<b>GL_LINEAR_MIPMAP_LINEAR</b>	Выбирает два уровня текстурной детализации наиболее близко соответствующих размеру пикселя. При этом выбор точек текстуры в каждом текстурном слое осуществляется по алгоритму <b>GL_LINEAR</b> . Итоговым значением текстуры является среднеарифметическое значение двух выбранных точек текстуры. На практике эту фильтрацию называют – трилинейной

#### 13.1.4 Взаимодействие текстуры с материалом объекта

Кроме задания внутренних параметров текстуры в OpenGL необходимо определить визуальное взаимодействие текстуры с видовыми атрибутами текстурируемого объекта. Для этого предусмотрена команды

```

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

```

Это многофункциональные команды с богатыми возможностями обработки текстур. Для полного изучения их возможностей смотрите в MSDN. В нашем случае ограничимся базовыми функциями.

Параметр *target* определяет цель действия функции **glTexEnv\*** и должен быть равен **GL\_TEXTURE\_ENV**.

Параметр *pname* задает имя параметра и будем использовать **GL\_TEXTURE\_ENV\_MODE**.

В качестве конкретных значений параметра будем использовать константы **GL\_MODULATE** и **GL\_REPLACE**.

В случае **GL\_MODULATE** итоговое значение текстурируемого участка будет определяться умножением значений точек участка на значение соответствующих точек из текстуры.

В случае **GL\_REPLACE** итоговое значение текстурируемого участка будет определяться только значением соответствующих точек из текстуры.

### 13.1.5 Координаты текстуры

В OpenGL координаты точек на текстуре представлены в параметрической системе координат (*s*, *t*). Значения *s* и *t* находятся в отрезке [0,1].

Для текстурирования объекта необходимо указать связь вершин объекта и соответствующих точек в текстуре (см. рис. 6). Для этой цели

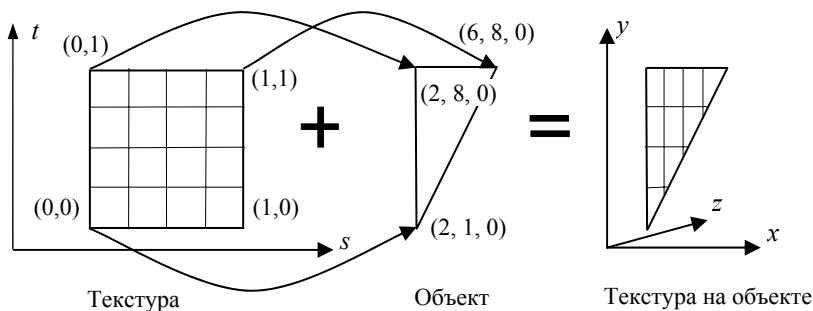


Рис. 6 Текстурирование объекта

имеются несколько команд. Одна из них в явном виде задает координаты текстуры для каждой вершины объекта и имеет вид:

```

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

```

Команда **glTexCoord2\*** устанавливает связь координат *s* и *t* с ассоциируемой вершиной объекта.

Например, для ситуации изображенной на рисунке фрагмент кода программы имеет вид

```

glBegin(GL_TRIANGLES); // Рисуем треугольник
glTexCoord2f(0.0f,1.0f);glVertex3f(2.0f,8.0f,0.0f);
glTexCoord2f(1.0f,1.0f);glVertex3f(6.0f,8.0f,0.0f);
glTexCoord2f(0.0f,0.0f);glVertex3f(2.0f,1.0f,0.0f);
glEnd();

```

Если требуется "размножение" текстуры на поверхности объекта, то это можно реализовать изменением кратности параметра *s* или *t* в команде **glTexCoord2\***. Например, нам необходимо уложить на текстурируемую поверхность 10 "картинок" текстуры по ширине. Для этого достаточно записать следующий код программы

```

glBegin(GL_QUADS); // Рисуем прямоугольник
glTexCoord2f( 0.0f,0.0f);glVertex3f( 1.0f,1.0f,1.0f);
glTexCoord2f( 0.0f,1.0f);glVertex3f( 1.0f,5.0f,1.0f);
glTexCoord2f(10.0f,1.0f);glVertex3f(11.0f,5.0f,1.0f);
glTexCoord2f(10.0f,0.0f);glVertex3f(11.0f,1.0f,1.0f);

```



glEnd();

Другой способ задания сопоставления координат текстуры с координатами вершин объекта основан на задании некоторой функции, которая в текущий момент времени рассчитывает координаты текстуры для каждой вершины объекта. И реализовано это командами

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

Параметр *coord* определяет координату текстуры, к которым будет применяться функция преобразования, и может принимать одно из значений GL\_S, GL\_T, GL\_R или GL\_Q.

Имя параметра *pname* должно принимать значение GL\_TEXTURE\_GEN\_MODE (для первой команды).

В качестве параметра *param* можно использовать следующие значения: GL\_OBJECT\_LINEAR, GL\_EYE\_LINEAR или GL\_SPHERE\_MAP.

Параметр	Описание
GL_OBJECT_LINEAR	<ul style="list-style-type: none"><li>• <b>СООТВЕТСТВУЮЩИЕ КООРДИНАТЫ <i>S</i>, <i>T</i>, <i>R</i> ИЛИ <i>Q</i> ВЫЧИСЛЯЮТСЯ ПО ФОРМУЛЕ</b> <math display="block">g = p_1x_0 + p_2x_0 + p_3z_0 + p_4w_0,</math><b>ГДЕ <math>p_1, \dots, p_4</math> – ЗНАЧЕНИЯ, НАХОДЯЩИЕСЯ В <i>PARAMS</i>; <math>x_0, \dots, w_0</math> – МИРОВЫЕ КООРДИНАТЫ ВЕРШИНЫ (СМ. MSDN)</b></li></ul>
GL_EYE_LINEAR	<ul style="list-style-type: none"><li>• Соответствующие координаты <i>s</i>, <i>t</i>, <i>r</i> или <i>q</i> вычисляются по формуле <math display="block">g = p_1x_e + p_2x_e + p_3z_e + p_4w_e,</math>где <math>(p_1', p_2', p_3', p_4') = (p_1, p_2, p_3, p_4)M^{-1}</math>; <i>M</i> – матрица видового преобразования; <math>x_e, \dots, w_e</math> – видовые координаты вершины. Координаты текстуры объекта в этом случае зависят от положения объекта (см. MSDN)</li></ul>
GL_SPHERE_MAP	<ul style="list-style-type: none"><li>• Дана функция позволяет моделировать эффект зеркального отражения от поверхности объекта. Для ее исполнения требуются видовые координаты и задание нормалей (см. MSDN)</li></ul>

Включение автоматического режима задания текстурных координат осуществляется командой glEnable с параметром GL\_TEXTURE\_GEN\_S или GL\_TEXTURE\_GEN\_P.

## 13.2 Работа со светом

OpenGL дает богатые возможности разработчику моделировать реалистическую графику сцен, где присутствует свет. Предусмотрен механизм задания световых характеристик материала объекта, параметров источников света и модели освещения. Рассмотрим эти возможности.

### 13.2.1 Задание свойств материала объекта

В OpenGL свойства материалов задаются командами

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

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

Здесь параметр *face* определяет лицевую или обратную поверхность, свойства материала которой необходимо изменить, и может принимать аргументы **GL\_FRONT** (лицевые), **GL\_BACK** (обратные) или **GL\_FRONT\_AND\_BACK** (обе стороны).

Аргумент *pname* определяет изменяемые параметры материала и может принимать следующие параметры.

Параметр	Описание
<b>GL_AMBIENT</b>	<ul style="list-style-type: none"> <li>• <b>ИЗМЕНЯЕТ ЦВЕТ РАССЕЙННОГО ОТРАЖЕНИЯ МАТЕРИАЛА. ПАРАМЕТР PARAM ДОЛЖЕН СОДЕРЖАТЬ ЧЕТЫРЕ ЦЕЛЫХ ИЛИ ВЕЩЕСТВЕННЫХ ЗНАЧЕНИЯ ЦВЕТА РАССЕЙННОГО ОТРАЖЕНИЯ (RGBA) МАТЕРИАЛА. ПО УМОЛЧАНИЮ ЗНАЧЕНИЕ ЦВЕТА РАССЕЙННОГО ОТРАЖЕНИЯ РАВНО (0.2, 0.2, 0.2, 1.0)</b></li> </ul>
<b>GL_DIFFUSE</b>	<ul style="list-style-type: none"> <li>• Изменяет цвет диффузного отражения материала. Параметр <i>param</i> должен содержать четыре целых или вещественных значения цвета диффузного отражения (RGBA) материала. По умолчанию значение цвета диффузного отражения равно (0.8, 0.8, 0.8, 1.0)</li> </ul>
<b>GL_SPECULAR</b>	<ul style="list-style-type: none"> <li>• Изменяет цвет зеркального отражения материала. Параметр <i>param</i> должен содержать четыре целых или вещественных значения цвета зеркального отражения (RGBA) материала. По умолчанию значение цвета зеркального отражения равно (0.0, 0.0, 0.0, 1.0)</li> </ul>
<b>GL_SHININESS</b>	<ul style="list-style-type: none"> <li>• Изменяет пространственное распределение зеркального отражения материала. Параметр <i>param</i> должен содержать одно целое или вещественное значение из диапазона ( 0, 128). По умолчанию степень зеркального отражения равен 0</li> </ul>

Параметр	Описание
<b>GL_EMISSION</b>	<ul style="list-style-type: none"> <li>Изменяет интенсивность излучаемого света материала. Параметр <i>param</i> должен содержать четыре целых или вещественных значения интенсивности излучаемого света (RGBA) материала. По умолчанию значение интенсивности излучаемого света равно (0.0, 0.0, 0.0, 1.0)</li> </ul>
<b>GL_AMBIENT_AND_DIFFUSE</b>	<ul style="list-style-type: none"> <li>Эквивалентно двум вызовам команды <code>glMaterial*()</code> со значением <i>pname</i> <b>GL_AMBIENT</b> и <b>GL_DIFFUSE</b> и одинаковыми значениями <i>params</i></li> </ul>

Изменять параметры материала можно и командой  
`void glColorMaterial (GLenum face, GLenum mode)`

Здесь *face* принимает аналогичные значения, что и в команде `glMaterial*()`, а *mode* –

**GL\_EMISSION**, **GL\_AMBIENT**, **GL\_DIFFUSE**, **GL\_SPECULAR** и **GL\_AMBIENT\_AND\_DIFFUSE**. По умолчанию используется **GL\_AMBIENT\_AND\_DIFFUSE**. Параметры материала поверхности, заданные аргументами *face* и *mode*, принимают значения текущего цвета (предварительно разрешив эту команду, вызвав `glEnable(GL_COLOR_MATERIAL)`). В такой ситуации можно изменять параметры материала для каждой вершины, используя только команду `glColor*`, что более удобно в некоторых случаях.

#### ПРИМЕР ЗАДАНИЯ ПАРАМЕТРОВ МАТЕРИАЛА:

```

...
float amb[] = {0.4, 0.3, 0.4}; // Цвет рассеянного отражения
float dif[] = {0.7, 0.4, 0.7}; // Цвет диффузного отражения
float spec[] = {0.5, 0.5, 0.2}; // Цвет зеркального отражения
float shininess = 90; // Степень зеркально отражения
...
// Установка свойств материала для лицевой поверхности
glMaterialfv (GL_FRONT, GL_AMBIENT, amb); // Для рассеянного света
glMaterialfv (GL_FRONT, GL_DIFFUSE, dif); // Для диффузного отражения.
glMaterialfv (GL_FRONT, GL_SPECULAR, spec); // Для зеркального отражения
// Степень зеркального отражения
glMaterialf (GL_FRONT, GL_SHININESS, shininess);
...

```

#### 13.2.2 Задание источников света

При работе с источником света необходимо определить его и присвоить ему порядковый номер. Для этой цели существует команда

```

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

```

Аргумент *light* закрепляет номер за источником света. В качестве номера выступает символическое имя `GL_LIGHTi`, где *i* может принимать значение из диапазона 0 до `GL_MAX_LIGHTS`. OpenGL поддерживает восемь источников света одновременно.

Для неекторной формы команды аргумент *pname* является символьной константой, определяющей устанавливаемый параметр:

Параметр	Описание
----------	----------

Параметр	Описание
GL_SPOT_EXPONENT	<ul style="list-style-type: none"> <li>ПАРАМЕТР <i>PARAM</i> ДОЛЖЕН СОДЕРЖАТЬ ЦЕЛОЕ ИЛИ ВЕЩЕСТВЕННОЕ ЗНАЧЕНИЕ ИЗ ДИАПАЗОНА [0, 128], КОТОРОЕ ЗАДАЕТ РАСПРЕДЕЛЕНИЕ ИНТЕНСИВНОСТИ ИЗЛУЧЕНИЯ СВЕТА. ЧЕМ БОЛЬШЕ ЭТО ЗНАЧЕНИЕ, ТЕМ БОЛЕЕ СФОКУСИРОВАН ИСТОЧНИК СВЕТА. ПО УМОЛЧАНИЮ ДАННЫЙ ПАРАМЕТР УСТАНОВЛЕН В ЗНАЧЕНИЕ 0, ЧТО СООТВЕТСТВУЕТ РАССЕЯННОМУ ОСВЕЩЕНИЮ</li> </ul>
GL_SPOT_CUTOFF	<ul style="list-style-type: none"> <li>Параметр <i>param</i> должен содержать целое или вещественное значение из диапазона [0, 90] или 180, которое задает угол максимальный угол разброса излучения света. Чем больше это значение, тем больше пятно освещения будет на поверхности объекта. По умолчанию данный параметр установлен в значение 180, что соответствует рассеянному освещению</li> </ul>
GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION	<ul style="list-style-type: none"> <li>Параметр <i>param</i> должен содержать целое или вещественное значение из диапазона [0, 90] или 180, которое задает один из трех факторов ослабления освещения – постоянного, линейного или квадратичного. Можно использовать только положительные значения. Интенсивность источника света ослабевает в согласно формуле <math display="block">\frac{1}{k_c + k_l d + k_q d^2}</math> где <math>k_c</math>, <math>k_l</math> и <math>k_q</math> – соответствующие коэффициенты ослабления, заданные соответствующими параметрами констант, </li> </ul>

Параметр	Описание
	$d$ – расстояние

Продолжение табл.

Параметр	Описание
<b>GL_CONSTANT_ATTENUATION,</b> <b>GL_LINEAR_ATTENUATION,</b> <b>GL_QUADRATIC_ATTENUATION</b>	между источником света и освещаемой им вершиной объекта. По умолчанию факторы ослабления имеют значения (1,0,0)

Для векторной формы команды **glLight[I f]v** добавляются дополнительные значения параметра *params*:

Параметр	Описание
<b>GL_AMBIENT</b>	<ul style="list-style-type: none"> <li>ОПРЕДЕЛЯЕТ ИНТЕНСИВНОСТЬ (ЦВЕТ) ФОНОВОГО ОСВЕЩЕНИЯ ОТ ДАННОГО ИСТОЧНИКА СВЕТА. ПАРАМЕТР <i>PARAMS</i> ДОЛЖЕН СОДЕРЖАТЬ ЧЕТЫРЕ ЦЕЛЫХ ИЛИ Вещественных числа RGBA, КОТОРЫЕ ОПРЕДЕЛЯЮТ ЗНАЧЕНИЕ ФОНОВОГО ОСВЕЩЕНИЯ. ПО УМОЛЧАНИЮ ЗНАЧЕНИЕ ФОНОВОГО ОСВЕЩЕНИЯ РАВНО (0.0, 0.0, 0.0, 1.0)</li> </ul>
<b>GL_DIFFUSE</b>	<ul style="list-style-type: none"> <li>Определяет интенсивность (цвет) диффузного освещения от данного источника света. Параметр <i>params</i> должен содержать четыре целых или вещественных числа RGBA, которые определяют значение диффузного освещения. По умолчанию значение диффузного освещения от нулевого источника света равно (1.0, 1.0, 1.0, 1.0), а от остальных (0.0, 0.0, 0.0, 1.0)</li> </ul>
<b>GL_SPECULAR</b>	<ul style="list-style-type: none"> <li>Определяет интенсивность (цвет) зеркального отражения от данного источника света. Параметр <i>params</i> должен содержать четыре целых или вещественных числа RGBA, которые определяют значение зеркального отражения. По умолчанию значение зеркального отражения от нулевого источ-</li> </ul>

Параметр	Описание
	ника света равно (1.0, 1.0, 1.0, 1.0), а от остальных (0.0, 0.0, 0.0, 1.0)

Продолжение табл.

Параметр	Описание
<b>GL_POSITION</b>	<ul style="list-style-type: none"> <li>Определяет положение источника света в мировых координатах. Параметр <i>params</i> должен содержать четыре целых или вещественных значения, определяющих положение источника света в однородных мировых координатах. Эта позиция преобразуется видовой матрицей и сохраняется в видовых координатах. Если компонент <i>w</i> положения равен 0.0, то свет рассматривается как направленный источник (расположенный в бесконечности), а диффузное и зеркальное освещение рассчитываются в зависимости от направления на источник света, но не от его действительного положения, и ослабление заблокировано. В противном случае эти параметры рассчитываются на основе действительного расположения источника в видовых координатах и ослабление разрешено. По умолчанию источник располагается в точке (0, 0, 1, 0), является направленным и световой поток параллелен оси <i>z</i></li> </ul>
<b>GL_SPOT_DIRECTION</b>	<ul style="list-style-type: none"> <li>Определяет положение источника света в однородных мировых координатах. Параметр <i>params</i> должен содержать четыре целых или вещественных значения, задающих направление света в однородных мировых координатах, интенсивности излучаемого света (RGBA) материала. По умолчанию направление источника света задается значениями (0.0, 0.0, -1.0, 1.0). Обработка этого параметра имеет смысл, если значение <b>GL_SPOT_CUTOFF</b> отличное от 180</li> </ul>

После того как установлены параметры источников света, эти источники можно включать в любое время. Предварительно требуется установить режим текущей освещенности вызовом команды `glEnable(GL_LIGHTING)`. Для включения *i*-го источника света используются команды `glEnable` с аргументами `GL_LIGHTi`. Выключение осуществляется командой `glDisable` с аналогичным аргументом.

**РАССМОТРИМ ПРИМЕР ОСВЕЩЕНИЯ СФЕРЫ ОДНИМ ИСТОЧНИКОМ СВЕТА:**

...

```

// Задаем исходные значения параметров материала сферы
GLfloat m_specular[]={1.0,1.0,1.0,1.0}; // Зеркальный цвет
GLfloat m_shininess[]={40.0}; // и его пространственное распределение
// Задаем исходные значения параметров источника света
GLfloat Ambient[]={0.5,0.5,0.5,1.0}; // Фоновое освещение
GLfloat Diffuse[]={1.0,1.0,1.0,1.0}; // Диффузное освещение
GLfloat Specular[]={1.0,1.0,1.0,1.0}; // Зеркальное отражение
GLfloat Position[]={1.0,1.0,1.0,1.0}; // Позиция света
...
// Процедура начальной установки
GLvoid Initial(GLsizei w, GLsizei h)
{
  Resize (w, h);
  glClearColor(0.0,0.0,0.0,0.0);
  // Устанавливаем параметры материала
  glMaterialfv(GL_FRONT,GL_SPECULAR,m_specular);
  glMaterialfv(GL_FRONT,GL_SHININESS,m_shininess);
  // Устанавливаем параметры источника света
  glLightfv(GL_LIGHT0,GL_POSITION, Position);
  glLightfv(GL_LIGHT0,GL_AMBIENT, Ambient);
  glLightfv(GL_LIGHT0,GL_DIFFUSE,Diffuse);
  glLightfv(GL_LIGHT0,GL_SPECULAR,Specular);
  // Устанавливаем параметры источника света
  glEnable(GL_LIGHTING); // Разрешение текущей освещенности
  glEnable(GL_LIGHT0); // Включение GL_LIGHT0 источник света
  glEnable(GL_DEPTH_TEST); // Включаем тест глубины
}
...
GLvoid DrawPrim()
{
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  quadratic=gluNewQuadric();
  gluSphere (quadratic,1 ,40 ,40); // Рисуем сферу
}

```

### 13.2.3 Задание модели освещения

Для задания свойств модели освещения используются команды

```

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

```

Аргумент *pname* определяет изменяемый параметр модели освещения и может принимать следующие значения в виде символьных констант (*param* – конкретное значение параметра).

<i>pname</i>	Значение
<b>GL_LIGHT_MODEL_LOCAL_VIEWER</b>	<ul style="list-style-type: none"> <li>ОПРЕДЕЛЯЕТ СПОСОБ ВЫЧИСЛЕНИЯ ЗЕРКАЛЬНОГО ОТРАЖЕНИЯ. ЕСЛИ <i>PARAM</i> = GL_FALSE, ТО</li> </ul>

<i>pname</i>	Значение
	<p><b>СЧИТАЕТСЯ ЧТО НАБЛЮДАТЕЛЬ РАСПОЛОЖЕН БЕСКОНЕЧНО ДАЛЕКО И НАПРАВЛЕНИЕ ОБЗОРА ПАРАЛЛЕЛЬНО ОСИ Z. ПРИ ЭТОМ ПОЛОЖЕНИЕ НАБЛЮДАТЕЛЯ НЕ ЗАВИСИТ ОТ РЕАЛЬНЫХ МИРОВЫХ КООРДИНАТ. В ПРОТИВНОМ СЛУЧАЕ, КОГДА <i>PARAM</i> = <b>GL_TRUE</b>, СЧИТАЕТСЯ, ЧТО НАБЛЮДАТЕЛЬ РАСПОЛОЖЕН В НАЧАЛЕ ВИДОВОЙ СИСТЕМЫ КООРДИНАТ. ПО УМОЛЧАНИЮ <i>PARAM</i> = <b>GL_TRUE</b></b></p>
<b>GL_LIGHT_MODEL_TWO_SIDE</b>	<ul style="list-style-type: none"> <li>• Определяет одностороннее или двухстороннее освещение. Если <i>param</i> = <b>GL_FALSE</b>, то рассчитывается освещение только для лицевых граней. В противном случае, когда <i>param</i> = <b>GL_TRUE</b>, задается двухстороннее освещение. Для обратных граней необходимо задавать свои свойства материала, нормали (противоположные нормальям лицевых граней). По умолчанию <i>param</i> = <b>GL_FALSE</b></li> </ul>
<b>GL_LIGHT_MODEL_AMBIENT</b> (только для векторной формы команды <code>glLightModel*v</code> )	<ul style="list-style-type: none"> <li>• Задает RGBA интенсивность фонового освещения. По умолчанию цвет фона равен (0.2, 0.2, 0.2, 1.0)</li> </ul>

### 13.3 Моделирование тумана

Для придания большей реалистичности в визуализации 3D сцен в OpenGL часто используют эффект тумана. Туман в OpenGL реализуется путем изменения цвета объектов в сцене в зависимости от расстояния до точки наблюдения.

Способ вычисления интенсивности тумана можно определить с помощью команд



```
void glFog[if] (
    GLenum pname,
    GLtype param)
```

```
void glFog[if]v (
    GLenum pname,
    const GLtype *params)
```

Аргумент *pname* может принимать следующие часто используемые значения:

Параметр	Описание
<b>GL_FOG_MODE</b>	<p>• ОПРЕДЕЛЯЕТ ФОРМУЛУ ПО КОТОРОЙ БУДЕТ ВЫЧИСЛЯТЬСЯ ИНТЕНСИВНОСТЬ ТУМАНА В ТОЧКЕ. АРГУМЕНТ <i>PARAM</i> МОЖЕТ ПРИНИМАТЬ ЗНАЧЕНИЯ:</p> <p>GL_EXP – ДЛЯ ВЫЧИСЛЕНИЯ ИНТЕНСИВНОСТИ ПО ФОРМУЛЕ <math>F = \text{EXP}(-D \cdot Z)</math>;</p> <p>GL_EXP2 – ДЛЯ ВЫЧИСЛЕНИЯ ИНТЕНСИВНОСТИ ПО ФОРМУЛЕ <math>F = \text{EXP}((-D \cdot Z)^2)</math>;</p> <p>GL_LINEAR – ДЛЯ ВЫЧИСЛЕНИЯ ИНТЕНСИВНОСТИ ПО ФОРМУЛЕ <math>F = E - Z / E - S</math>, ГДЕ <i>Z</i> – РАССТОЯНИЕ ОТ ВЕРШИНЫ, В КОТОРОЙ ВЫЧИСЛЯЕТСЯ ИНТЕНСИВНОСТЬ ТУМАНА, ДО ТОЧКИ НАБЛЮДЕНИЯ.</p> <p>КОЭФФИЦИЕНТЫ <i>D</i>, <i>S</i>, <i>E</i> ЗАДАЮТСЯ С ПОМОЩЬЮ СЛЕДУЮЩИХ ЗНАЧЕНИЙ АРГУМЕНТА <i>PNAME</i>: GL_FOG_DENSITY, GL_FOG_START И GL_FOG_END СООТВЕТСТВЕННО</p>
<b>GL_FOG_DENSITY</b>	<p>• Плотность тумана (<i>d</i>) для GL_EXP и GL_EXP2. <i>param</i> определяет значение <i>d</i>. По умолчанию установлено значение (1.0)</p>
<b>GL_FOG_START</b>	<p>• Начальное расстояние (<i>s</i>) от точки наблюдения после которого начинает изменяться интенсивность тумана. <i>param</i> определяет значение <i>s</i>. По умолчанию установлено значение (0.0)</p>

Продолжение табл.

Параметр	Описание
----------	----------

Параметр	Описание
<b>GL_FOG_END</b>	<ul style="list-style-type: none"> <li>Конечное расстояние (<math>e</math>) от точки наблюдения до которого изменяется интенсивность тумана. <i>param</i> определяет значение <math>e</math>. По умолчанию установлено значение (1.0)</li> </ul>
<b>GL_FOG_COLOR</b>	<ul style="list-style-type: none"> <li>Определяется цвет тумана. В этом случае <i>params</i> – указатель на массив из 4-х компонент цвета (RGBA). По умолчанию установлено значение (0.0, 0.0, 0.0, 0.0)</li> </ul>

### РАССМОТРИМ ПРИМЕР ИСПОЛЬЗОВАНИЯ ТУМАНА:

```

...
GLfloat FogColor[4]={0.6,0.6,0.6,1.0}; // Цвет тумана
...
glEnable(GL_FOG); // Включаем туман
glFogi(GL_FOG_MODE, GL_LINEAR); // Линейное изменение интенсивности
glFogf(GL_FOG_START, 2.0); // Начало тумана
glFogf(GL_FOG_END, 50.0); // Окончание тумана
glFogfv(GL_FOG_COLOR, FogColor); // Устанавливаем цвет тумана
...

```

#### 13.4 Прозрачность

Реалистическая графика предполагает использование эффекта прозрачности элементов наблюдаемой сцены, когда через одни фрагменты сцены можно видеть другие. Для этого цели введено понятие альфа-канал в характеристике цвета точки (RGBA), который и указывает на степень ее "прозрачности". При альфа равном 1.0 фрагмент считается полностью непрозрачным, а при 0.0 – полностью прозрачным.

Тест на прозрачность разрешается и запрещается командами `glEnable` и `glDisable` с параметром **GL\_ALPHA\_TEST**.

Для управления тестом по альфа-каналу используется команда  
**void glAlphaFunc**(GLenum *func*, GLclampf *ref*),

где параметр *ref* задает сравниваемое значение для альфа-параметра. Это значение находится в диапазоне [0,1]. По умолчанию параметр *ref* равен 0. Параметр *func* задает функцию сравнения значений альфа и может принимать следующие символьные константы:

Параметр	Тест завершается положительно в случае
<b>GL_NEVER</b>	<b>НИКОГДА</b>
<b>GL_LESS</b>	Если поступающее значение меньше, чем <i>ref</i>
<b>GL_EQUAL</b>	Если поступающее значение равно <i>ref</i>
<b>GL_LEQUAL</b>	Если поступающее значение меньше или равно <i>ref</i>
<b>GL_GREATER</b>	Если поступающее значение больше, чем <i>ref</i>
<b>GL_NOTEQUAL</b>	Если поступающее значение не равно <i>ref</i>

Параметр	Тест завершается положительно в случае
<b>GL_GEQUAL</b>	Если поступающее значение больше или равно <i>ref</i>
<b>GL_ALWAYS</b>	Всегда. Установлено по умолчанию

Данная команда позволяет принять или отклонить фрагмент, основываясь на сравнении заданного значения альфа-канала с реальным. Если результат сравнения положительный, то поступающий фрагмент рисуется в буфере кадра (в зависимости от условий тестов трафарета и глубины). В случае отрицательного результата – вывод фрагмента не осуществляется.

В данном примере выводится фрагмент объекта в буфер кадра, если его альфа-значение больше (0.4):

```
glEnable(GL_ALPHA_TEST);
glAlphaFunc(GL_GREATER, 0.4f);
```

Используя команду `glAlphaFunc()` мы работаем с полностью прозрачным или полностью непрозрачным смешиванием, но при этом не возникает реального смешивания. Для определения метода реального смешивания выводимых фрагментов с изображением существующим в буфере кадра используется команда

```
void glBlendFunc(GLenum srcfactor, GLenum dstfactor)
```

Параметр *srcfactor* определяет метод обработки RGBA значений, поступающих от источника на смешивание (выводимый фрагмент), а *dstfactor* определяет метод обработки RGBA значений, находящихся в буфере кадра с которыми будет осуществляться смешивание. Всего существует одиннадцать методов обработки RGBA значений. Девять из них предназначены для источника (*srcfactor*) и восемь для приемника (*dstfactor*).

Параметр	Метод
<b>GL_ZERO</b>	<b>(0, 0, 0, 0)</b>
<b>GL_ONE</b>	(1, 1, 1, 1)
<b>GL_SRC_COLOR</b>	$(R_S / k_R, G_S / k_G, B_S / k_B, A_S / k_A)$

Продолжение табл.

Параметр	Метод
<b>GL_ONE_MINUS_SRC_COLOR</b>	$(1, 1, 1, 1) - (R_S / k_R, G_S / k_G, B_S / k_B, A_S / k_A)$
<b>GL_DST_COLOR</b>	$(R_D / k_R, G_D / k_G, B_D / k_B, A_D / k_A)$
<b>GL_ONE_MINUS_DST_COLOR</b>	$(1, 1, 1, 1) - (R_D / k_R, G_D / k_G, B_D / k_B, A_D / k_A)$
<b>GL_SRC_ALPHA</b>	$(A_S / k_A, A_S / k_A, A_S / k_A, A_S / k_A)$
<b>GL_ONE_MINUS_SRC_ALPHA</b>	$(1, 1, 1, 1) - (A_S / k_A, A_S / k_A, A_S / k_A, A_S / k_A)$
<b>GL_DST_ALPHA</b>	$(A_D / k_A, A_D / k_A, A_D / k_A, A_D / k_A)$
<b>GL_ONE_MINUS_DST_ALPHA</b>	$(1, 1, 1, 1) - (A_D / k_A, A_D / k_A, A_D / k_A, A_D / k_A)$

Параметр	Метод
<b>GL_SRC_ALPHA_SATURATE</b>	$(i, i, i, 1)$ , где $i = \min(A_S, k_A - A_D / k_A)$

В таблице приняты следующие условные обозначения:  $R_S, G_S, B_S, A_S$  определяют составляющие цвета источника;  $R_D, G_D, B_D, A_D$  определяют составляющие цвета приемника. Значение цвета изменяется в диапазоне  $[0, k_C = 2^{m_C} - 1]$ , где  $C$  означает составляющие цвета  $R, G, B$  и  $A$ , а  $m$  – количество битовых плоскостей видео памяти, отведенных под  $R, G, B$  и  $A$  соответственно. Для определения итогового значения RGBA после смешивания используются следующие уравнения:

$$R(d) = \min(k_R, R_S \cdot s_R + R_D \cdot d_R);$$

$$G(d) = \min(k_G, G_S \cdot s_G + G_D \cdot d_G);$$

$$B(d) = \min(k_B, B_S \cdot s_B + B_D \cdot d_B);$$

$$A(d) = \min(k_A, A_S \cdot s_A + A_D \cdot d_A),$$

где  $s_R, s_G, s_B, s_A, d_R, d_G, d_B, d_A$  – соответствующие компоненты векторов методов из таблицы для источника и для приемника цвета.

По умолчанию смешивание заблокировано. Для его включения используется команда `glEnable(GL_BLEND)`. Для отключения – `glDisable(GL_BLEND)`.

Приведенные возможности смешивания цветов позволяют осуществить достаточно сложные замыслы разработчиков. На практике часто используют команду `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` с примитивами, отсортированными по глубине от наиболее к наименее отдаленным и непрозрачными элементами в глубине. Последнее подсказано логикой визуализации "пакета" полупрозрачных графических примитивов.

Рассмотрим пример вывода трех примитивов: непрозрачного красного треугольника; полупрозрачного зеленого квадрата и полупрозрачного синего прямоугольника. Процедура вывода имеет вид:

```
GLvoid DrawPrim()
{
glClear( GL_COLOR_BUFFER_BIT);
// Непрозрачный красный треугольник
glColor4f(1.0f, 0.0f,0.0f, 1.0f);
glBegin(GL_TRIANGLES);
glVertex3f( 0.0f, 1.8f, 1.0f);
glVertex3f(-1.4f, -1.5f, 1.0f);
glVertex3f( 1.4f, -1.5f, 1.0f);
glEnd();
glEnable(GL_BLEND); // Разрешаем смешивание цветов
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Режим смешивания
glColor4f(0.0f, 1.0f,0.0f, 0.5f); // Полупрозрачный зеленый квадрат
glBegin(GL_QUADS);
glVertex3f( 1.0f, 1.0f, 2.0f);
glVertex3f(-1.0f, 1.0f, 2.0f);
glVertex3f(-1.0f, -1.0f, 2.0f);
glVertex3f( 1.0f, -1.0f, 2.0f);
glEnd();
// Полупрозрачный синий прямоугольник
glColor4f(0.0f, 0.0f,1.0f, 0.5f);
glBegin(GL_QUADS);
glVertex3f( 2.0f, 0.5f, 0.0f);
glVertex3f(-2.0f, 0.5f, 0.0f);
glVertex3f(-2.0f, -0.5f, 0.0f);
```

```

glVertex3f( 2.0f, -0.5f, 0.0f);
glEnd();
glDisable(GL_BLEND); // Запрещаем смешивание цветов
}

```

В результате работы данной процедуры мы можем видеть красный треугольник через полупрозрачный квадрат и прямоугольник.

## 14 РАБОТА С БУФЕРАМИ

За общепринятым понятием буфер кадра в OpenGL скрывается не один, а несколько буферов – цвета, трафарета, глубины и аккумулятора (накопления). Это разделение позволяет расширить функциональные возможности OpenGL. Рассмотрим механизмы работы некоторых буферов.

### 14.1 Буфер трафарета

Буфер трафарета предоставляет огромные возможности для творчества. С его помощью реализуются самые разнообразные эффекты, начиная от простого вырезания одной фигуры из другой до реализации теней, отражений и прочих непростых функций, требующих от вас уже не только знакомство с библиотекой OpenGL, но и понимания алгоритмов машинной графики.

Наложение трафарета разрешает или нет рисование на пиксельном уровне. В качестве трафарета может выступать любой графический примитив. При этом рисование осуществляется в плоскости трафарета. После чего этот буфер используется как маска для вывода графики на экран. Главный механизм работы с трафаретом базируется на тестах трафарета, которые сравнивают значения в буфере трафарета с заданным значением. Если тест проходит, то тестируемый пиксель помещается в буфер кадра для визуализации.

Сам тест разрешается или блокируется командами `glEnable/glDisable` с аргументом `GL_STENCIL_TEST`. Очищается буфер трафарета при помощи функции `glClear` с параметром `GL_STENCIL_BUFFER_BIT`.

Для проведения трафаретного тестирования предусмотрены команды: `glStencilFunc`, отвечающая за функцию сравнения, и `glStencilOp`, определяющая действие на базе проверки трафарета.

Рассмотрим эти команды подробнее:

```
void glStencilFunc (GLenum func, int ref, GLuint mask)
```

Данная команда задает правило (параметром *func*), по которому будет определяться результат сравнения значений, хранящихся в буфере трафаретов (*stencil*) с некоторым заданным через параметр *ref*. Сравнение осуществляется по маске *mask*. Переменная *func* может принимать одно из следующих значений:

Константа	Тест ...
<code>GL_NEVER</code>	<b>НЕ ПРОХОДИТ НИКОГДА</b>
<code>GL_LESS</code>	Проходит, если $(ref \& mask) < (stencil \& mask)$
<code>GL_EQUAL</code>	Проходит, если $(ref \& mask) = (stencil \& mask)$
<code>GL_LEQUAL</code>	Проходит, если $(ref \& mask) \leq (stencil \& mask)$
<code>GL_GREATER</code>	Проходит, если $(ref \& mask) > (stencil \& mask)$
<code>GL_NOTEQUAL</code>	Проходит, если $(ref \& mask) \neq (stencil \& mask)$
<code>GL_GEQUAL</code>	Проходит, если $(ref \& mask) \geq (stencil \& mask)$
<code>GL_ALWAYS</code>	Проходит всегда

Параметр *ref* задает значение для сравнения и может принимать любое из диапазона  $[0, 2^n - 1]$ , где  $n$  – число битовых плоскостей в буфере трафаретов. Аргумент *mask* задает маску для значений *ref* и *stencil*.

По умолчанию тест трафарета заблокирован. В этом случае модификация буфера трафарета не может произойти и это означает положительный результат теста трафарета.

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

```
void glStencilOp (GLenum fail, GLenum zfail, GLenum zpass)
```

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

Константа	Действие
<b>GL_KEEP</b>	<b>СОХРАНИТЬ ТЕКУЩЕЕ ЗНАЧЕНИЕ В БУФЕРЕ ТРАФАРЕТА</b>
<b>GL_ZERO</b>	Установить значение буфера трафарета в ноль
<b>GL_REPLACE</b>	Заменить значение буфера трафарета на значение переменной <i>ref</i> , заданной командой <code>glStencilOp</code>
<b>GL_INCR</b>	Увеличить на единицу текущее значение буфера трафарета
<b>GL_DECR</b>	Уменьшить на единицу текущее значение буфера трафарета
<b>GL_INVERT</b>	Поразрядно инвертировать текущее значение трафарета

Параметры *zfail* и *zpass* определяют действия для отрицательного и положительного, соответственно, завершения теста глубины, которые будут выполнены в случае положительного результата выполнения теста трафарета. Для этих параметров допускаются те же символьные константы, что для параметра *fail*.

Рассмотрим простой пример использования буфера трафарета.

Пусть на фоне градиентно-закрашенного квадрата необходимо вывести зеленый треугольник с вырезанным окном в виде треугольника меньшего размера, через который можно видеть базовый квадрат. Квадрат расположен чуть дальше, чем остальные объекты. Рассмотрим поэтапно наши действия:

- 1 Определяем значение буфера трафарета для фона, например – 1.
- 2 Рисуем градиентно-закрашенный квадрат.
- 3 Заполняем в буфере трафаретов треугольное "окно" значениями, например – 2.
- 4 Выводим зеленого треугольника в тех местах, где значения в буфере трафарета не равны 2.

Рассмотрим фрагмент кода программы, выполняющий поставленную задачу. Комментарии подробно поясняют все выполняемые действия.

```
GLvoid DrawPrim()
{
// Значение, которым будет заполняться буфер стенсила при его очистке
glClearStencil(1);
// Определяем, какие биты могут писаться в стенсила буфер
glStencilMask(0xFF);
// Очищаем все буферы
glClear( GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glBegin(GL_QUADS); // Рисуем градиентно-закрашенный квадрат
glColor3f( 1,0,0);glVertex3f( 2.0f, 2.0f, -0.1f);
glColor3f( 0,1,0);glVertex3f(-2.0f, 2.0f, -0.1f);
glColor3f( 0,0,1);glVertex3f(-2.0f, -2.0f, -0.1f);
```

```

glColor3f( 1,0,1);glVertex3f( 2.0f, -2.0f, -0.1f);
glEnd();
// Разрешаем проведение теста трафарета для треугольного "окна"
glEnable(GL_STENCIL_TEST);
// Определяем функцию и значение для сравнения,
// треугольник не рисуется
glStencilFunc(GL_NEVER, 2, 0);
// Заменяем значение буфера трафарета на значение 2
glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP);
glBegin(GL_POLYGON); // Определяем область "окна"
glVertex3f( 0.0f, 1.0f, 0.0f);
glVertex3f( 1.0f, -1.0f, 0.0f);
glVertex3f(-1.0f, -1.0f, 0.0f);
glEnd();
// Изменяем параметры теста буфера трафарета;
// следующий объект будет рисоваться только в тех местах,
// для которых значение в буфере трафарета не равно 2
glStencilFunc(GL_NOTEQUAL, 2, 255);
glColor3f(0.0f, 1.0f,0.0f); // Задаем зеленый цвет
glBegin(GL_POLYGON); // Рисуем треугольник
glVertex3f( 0.0f, 1.6f, 0.0f);
glVertex3f(-1.4f, -1.4f, 0.0f);
glVertex3f( 1.4f, -1.4f, 0.0f);
glEnd();
}

```

При работе с буфером трафарета необходимо аккуратно продумывать последовательность действий.

## 14.2 Буферы накопления

В OpenGL возможно существование специально отведенных внутренних буферов (передний – **GL\_FRONT**, задний – **GL\_BACK**, левый – **GL\_LEFT**, правый – **GL\_RIGHT**, передний левый – **GL\_FRONT\_LEFT**, передний правый – **GL\_FRONT\_RIGHT**, задний левый – **GL\_BACK\_LEFT**, задний правый – **GL\_BACK\_RIGHT** и вспомогательные – **GL\_AUX0**, **GL\_AUX1**, **GL\_AUX2**, **GL\_AUX3**) для временного хранения визуальных изображений. Конкретное наличие того или иного буфера зависит от конфигурирования системы. На практике используют **GL\_FRONT** и **GL\_BACK**. При этом возможен процесс накапливания значений пиксель изображений и процесс считывания этих данных.

Выбор буфера для записи изображения, например, для работы команды `glDrawPixels()`, выполняется командой

```
void glDrawBuffer (GLenum mode)
```

Параметр *mode* определяет буфер для записи. Этой командой можно выбрать сразу несколько буферов и записывать изображение сразу в несколько буферов. По умолчанию *mode* равно **GL\_FRONT** для режима

с однократной буферизацией и **GL\_BACK** для режима с двойной буферизацией.

Для выбора цветового буфера, который будет служить в качестве источника для чтения пикселей, например, при вызове команд `glReadPixels()`, `glCopyPixels()`, `glCopyTexImage*()`, существует команда

```
void glReadBuffer (GLenum mode)
```

Параметр *mode* определяет буфер для чтения. Буферы для `glReadBuffers()` те же самые, что и для команды `glDrawBuffers()`. По умолчанию *mode* равно **GL\_FRONT** для режима с однократной буферизацией и **GL\_BACK** для режима с двойной буферизацией.

### 14.3 Буфер аккумулятора

В OpenGL существует специально отведенный внутренний буфер для временного хранения визуальных изображений. При этом возможен процесс накапливания значений пиксель изображений. Кроме этого, имеется возможность попиксельных операций над этими изображениями. На практике, подобные буферы используют для получения эффектов нерезкости, сглаживания, "мягких" теней и т.п.

Операции с буфером накопления осуществляются командой  
`void glAccum (GLenum op, GLfloat value)`

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

Константа	Операция над пикселями
<b>GL_LOAD</b>	<b>ПИКСЕЛИ БЕРУТСЯ ИЗ БУФЕРА, ВЫБРАННОГО НА ЧТЕНИЕ КОМАНДОЙ GLREADBUFFER(), ИХ ЗНАЧЕНИЯ УМНОЖАЮТСЯ НА VALUE И ЗАНОСЯТСЯ В БУФЕР-АККУМУЛЯТОР</b>
<b>GL_ACCUM</b>	Аналогично <b>GL_LOAD</b> , только результат не просто записывается в буфер-аккумулятор, а складывается с уже имеющимся в буфере
<b>GL_MULT</b>	Пиксели в буфере аккумулятора умножаются на значение переменной <i>value</i>
<b>GL_ADD</b>	Пиксели в буфере аккумулятора складываются со значением переменной <i>value</i>
<b>GL_RETURN</b>	Изображение переносится из буфера аккумулятора в буфер, выбранный для записи. Перед этим значение каждого пикселя умножается на <i>value</i>

Например, чтобы "смазать" изображение, в буфере аккумулятора одна и та же сцена рисуется несколько раз. Каждый раз с немного измененными значениями координат расположения камеры.

Буфер аккумулятора можно "очищать" определенными значениями составляющих цвета R, G, B и A. Для этого используется команда:

`void glClearAccum (GLfloat R, GLfloat G, GLfloat B, GLfloat alpha)`

### 14.4 Буфер глубины

Работая с трехмерной графикой нам приходится постоянно сталкиваться с глубиной сцены. На основании глубины работает алгоритм, использующий Z-буфер, для удаления невидимых граней. Каждое поступающее значение глубины фрагмента сравнивается с уже имеющимся в буфере глубины и выводится на экран (или нет) в зависимости от результатов выполнения этого теста.

По умолчанию тест глубины заблокирован. Для его включения/выключения используется команда `glEnable/glDisable` с аргументом **GL\_DEPTH\_TEST**. Функция сравнения, используемая в тесте глубины, задается командой

`void glDepthFunc (GLenum func)`

Данная команда определяет функцию сравнения для поступающего z-значения с тем, которое было в буфере глубины. Функция сравнения задается параметром *func*, которое может принимать следующие символьные значения:

Константа	Тест завершается положительно...
<b>GL_NEVER</b>	<b>НИКОГДА</b>
<b>GL_LESS</b>	Если поступающее z-значение меньше, чем хранящееся в буфере глубины



Константа	Тест завершается положительно...
<b>GL_EQUAL</b>	Если поступающее <i>z</i> -значение равно хранящемуся в буфере глубины
<b>GL_LEQUAL</b>	Если поступающее <i>z</i> -значение меньше или равно, чем хранящееся в буфере глубины
<b>GL_GREATER</b>	Если поступающее <i>z</i> -значение больше, чем хранящееся в буфере глубины
<b>GL_NOTEQUAL</b>	Если поступающее <i>z</i> -значение не равно хранящемуся в буфере глубины
<b>GL_GEQUAL</b>	Если поступающее <i>z</i> -значение больше или равно, чем хранящееся в буфере глубины
<b>GL_ALWAYS</b>	Всегда

По умолчанию функция сравнения задана как **GL\_LESS**.

#### 14.5 Буфер изображения

Для копирования цветного изображения из буфера изображения в обычную оперативную память используется команда:

```
void glReadPixels (GLint x, GLint y,
                  GLsizei width, GLsizei height,
                  GLenum format, GLenum type, GLvoid *pixels)
```

Параметры *x* и *y* задают левый нижний угол копируемого изображения, ширина которого *width* и высота *height*. Параметром *format* можно указать, что именно от пикселей необходимо копировать. Для этого предусмотрены следующие символьные константы: **GL\_RED**, **GL\_GREEN**, **GL\_BLUE**, **GL\_ALPHA**, **GL\_RGB**, **GL\_RGBA**, **GL\_COLOR\_INDEX**, **GL\_STENCIL\_INDEX**, **GL\_LUMINANCE**, **GL\_DEPTH\_COMPONENT** и т.д. (см. MSDN). Параметр *type* задает тип записываемых значений и может принимать следующие значения: **GL\_UNSIGNED\_BYTE**, **GL\_BYTE**, **GL\_BITMAP**, **GL\_UNSIGNED\_SHORT**, **GL\_SHORT**, **GL\_FLOAT**, **GL\_INT** и **GL\_UNSIGNED\_INT**. Место для размещения изображения в оперативной памяти указывается параметром *pixels*.

Для записи изображения из оперативной памяти в буфер изображения используется команда

```
void glDrawPixels (GLsizei width, GLsizei height,
                  GLenum format, GLenum type, GLvoid *pixels)
```

После отработки данной команды изображение должно появиться начиная с текущей позиции raster. Позицию можно задать, например, командой `glRasterPos()`.

Для копирования цветного изображения из одного участка буфера в другой участок, минуя оперативную память, используется команда

```
void glCopyPixels (GLint x, GLint y,
                  GLsizei width, GLsizei height, GLenum format)
```

Параметры *x* и *y* задают левый нижний угол копируемого изображения, ширина которого *width* и высота *height*. Параметром *format* можно указать, что именно от пикселей необходимо копировать (см. команду `glReadPixels()`). Позицию для размещения фрагмента изображения можно задать, например, командой `glRasterPos()`.

### 15 ГРУППЫ АТТРИБУТОВ

Работая с OpenGL разработчик вынужден постоянно изменять некоторые параметры примитивов графической библиотеки. Для того чтобы изменение параметров не влияло на состояние других примитивов существуют команды для сохранения и восстановления текущих параметров примитивов.

```
void glPushAttrib (GLbitfield mask)
void glPopAttrib ()
```

Первая команда сохраняет в стеке все атрибуты, указанные битами в параметре *mask*. `glPopAttrib()` восстанавливает значения тех переменных состояния, которые были сохранены командой `glPushAttrib()`. Возможные значения параметра *mask*:

<i>mask</i>	Сохраняемые атрибуты
<b>GL_ACCUM_BUFFER_BIT</b>	<b>ДЛЯ БУФЕРА НАКОПЛЕНИЯ (АККУМУЛЯТОРА). АККУМУЛЯТОР ОЧИЩАЕТСЯ</b>
<b>GL_ALL_ATTRIB_BITS</b>	Все доступные (!)
<b>GL_COLOR_BUFFER_BIT</b>	GL_ALPHA_TEST, GL_DRAW_BUFFER, GL_BLEND, GL_DITHER, GL_LOGIC_OP
<b>GL_CURRENT_BIT</b>	Текущие значения. Текущий RGBA цвет, текущий индекс цвета, вектора нормали, текущие координаты текстуры, текущая позиция растра, флаг доступности текущей позиции растра – GL_CURRENT_RASTER_POSITION_VALID

*Продолжение табл.*

<i>mask</i>	Сохраняемые атрибуты
<b>GL_DEPTH_BUFFER_BIT</b>	Биты тестирования буфера глубины – GL_DEPTH_TEST, маска записи глубины – GL_DEPTH_WRITEMASK, тестируемая функция глубины. Буфер глубины очищается.
<b>GL_ENABLE_BIT</b>	Все включенные атрибуты
<b>GL_EVAL_BIT</b>	Все вычислители
<b>GL_FOG_BIT</b>	Атрибуты тумана
<b>GL_HINT_BIT</b>	Параметры контролируемых задач (см. glHint)
<b>GL_LIGHTING_BIT</b>	Атрибуты освещения
<b>GL_LINE_BIT</b>	Атрибуты линии
<b>GL_LIST_BIT</b>	Стартовое смещение имен списков
<b>GL_PIXEL_MODE_BIT</b>	Атрибуты пиксель
<b>GL_POINT_BIT</b>	Атрибуты точки. Флаг GL_POINT_SMOOTH и размер точки
<b>GL_POLYGON_BIT</b>	Атрибуты полигонов
<b>GL_POLYGON_STIPPLE_BIT</b>	Атрибуты шаблона полигона
<b>GL_SCISSOR_BIT</b>	Атрибуты отсечения. Флаг GL_SCISSOR_TEST и поле отсе-

<i>mask</i>	Сохраняемые атрибуты
	чения
<b>GL_STENCIL_BUFFER_BIT</b>	Атрибуты для буфера трафаретов
<b>GL_TEXTURE_BIT</b>	Атрибуты текстуры
<b>GL_TRANSFORM_BIT</b>	Атрибуты преобразований
<b>GL_VIEWPORT_BIT</b>	Атрибуты просмотра

Более подробная информация параметров команд сохранения атрибутов находится в справочной системе (MSDN).

Для сохранения нескольких атрибутов необходимо для каждого атрибута выполнить команду сохранения `glPushAttrib()` (для восстановления атрибутов не забудьте исполнить столько же раз команду `glPopAttrib()` в обратном порядке).

### 16 УПРАВЛЕНИЕ ЭФФЕКТИВНОСТЬЮ РАБОТЫ АЛГОРИТМА РАСТЕРИЗАЦИИ

Иногда для повышения эффективности работы графического приложения можно упростить работу некоторых алгоритмов растеризации, например, при сглаживании точек и линий, при формировании тумана, при интерполировании координат для цвета и текстур и т.п. Правда, на это можно пойти, если нет нарушения эстетического восприятия итогового изображения. Управлять процессом растеризации можно командой

```
void glHint (GLenum target, GLenum mode)
```

Здесь *target* – вид контролируемой задачи, принимает одно из следующих значений:

<i>target</i>	Контролируемые параметры
<b>GL_FOG_HINT</b>	<b>ТОЧНОСТЬ ВЫЧИСЛЕНИЯ ТУМАНА</b>
<b>GL_POINT_SMOOTH_HINT</b>	Качество сглаживания (антиалиасинг) при растеризации точки
<b>GL_LINE_SMOOTH_HINT</b>	Качество сглаживания (антиалиасинг) при растеризации линии
<b>GL_POLYGON_SMOOTH_HINT</b>	Качество сглаживания (антиалиасинг) при растеризации сторон полигона
<b>GL_PERSPECTIVE_CORRECTION_HINT</b>	Процесс интерполяции значений координат точек при вычислении цвета и текстурировании

Возможные значения параметра *mode* в виде символьных констант:

<i>mode</i>	Значения
-------------	----------

<i>mode</i>	Значения
<b>GL_FASTEST</b>	<b>БЫСТРЫЙ АЛГОРИТМ</b>
<b>GL_NICETESR</b>	Алгоритм с высоким качеством результата
<b>GL_DONT_CARE</b>	Базовый алгоритм

OpenGL может отклонить ваши рекомендации, записанные в команде `glHint()`, из-за невозможности выполнения данного режима, так как конкретные реализации OpenGL могут по разному выполнять свои команды.

## ЗАКЛЮЧЕНИЕ

В узких рамках данного пособия трудно раскрыть все возможности OpenGL. Но мне кажется, что тот материал, который представлен, послужит хорошей базой для самостоятельного изучения и исследования возможностей библиотеки.

В пособие не вошли вопросы реалистической графики, связанные с построением теней и зеркального отражения. Дело в том, что OpenGL не располагает специальными средствами для получения подобных эффектов. Обычно используют индивидуальные способы построения теней и зеркального отражения. В частности, интересные алгоритмы построения теней различной сложности и на различные поверхности можно взять, например, с сайтов <http://ixbt.com> и <http://www.gamedev.ru>.

За очевидной мощью графической библиотеки скрываются множество "подводных камней", связанных с неэффективным использованием некоторых ее команд. Рассмотрим некоторые из них.

- Не увлекайтесь матричными операциями. Если, например, требуется для каждого кадра анимации осуществлять поворот `glRotate*` и перемещение `glTranslate*`, то гораздо эффективней две эти матрицы свести в одну и выполнять единственную эту матрицу используя команду `glLoadMatrixa*`.
- Старайтесь работать списками изображений, так как при очередном вызове дисплейного списка координаты вершин и их атрибуты уже находятся в памяти видеокарты, что положительно сказывается на производительности обработки изображения.
- Используйте векторную форму задания параметров команд OpenGL, что эффективней по сравнению скалярного задания соответствующих величин.

## СПИСОК ЛИТЕРАТУРЫ

- 1 **ТИХОМИРОВ, Ю. ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ // Ю. ТИХОМИРОВ. СПБ.: ВНУ, 1998.**
- 2 Баяковский, Ю.М. Графическая библиотека OpenGL: учебно-методическое пособие // Ю.М. Баяковский, А.В. Игнатенко, А.И. Фролов. М.: ВМиК МГУ, 2003.
- 3 **ПОРЕВ, В.Н. КОМПЬЮТЕРНАЯ ГРАФИКА // В.Н. ПОРЕВ. СПБ.: ВНУ, 2002.**
- 4 Шикин, Е.В. Компьютерная графика. Полигональные модели // Е.В. Шикин, А.В. Боресков. М.: Диалог МИФИ, 2000.

## ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Команды GL  
`glAccum`, 70  
`glAlphaFunc`, 63  
`glArrayElement`, 26  
`glBegin`, 9  
`glBitmap`, 18  
`glBlendFunc`, 64  
`glCallList`, 28  
`glClear`, 7  
`glClearAccum`, 71

`glClearColor`, 7  
`glClipPlane`, 21  
`glColor`, 8  
`glColorMaterial`, 56  
`glColorPointer`, 25  
`glCullFace`, 16  
`glDeleteList`, 30  
`glDepthFunc`, 71  
`glDepthRange`, 40  
`glDisableClientState`, 26

glDrawArrays, 26  
 glDrawBuffer, 70  
 glDrawPixels, 72  
 glEnableClientState, 25  
 glEnd, 9  
 glEndList, 27  
 glFinish, 6  
 glFog, 62  
 glFrontFace, 16  
 glFrustum, 41  
 glGenList, 30  
 glGetDoublev, 44  
 glGetIntegerv, 44  
 glHint, 75  
 glLight, 56  
 glLightModel, 60  
 glLinesStipple, 15  
 glLineWidth, 15  
 glListBase, 30  
 glLoadIdentity, 36  
 glLoadMatrix, 35  
 glLogicOp, 22  
 glMaterial, 54  
 glMatrixMode, 35  
 glMultMatrix, 36  
 glNewList, 27  
 glNormal3, 14  
 glNormalPointer, 25  
 glOrtho, 40  
 glPixelStore, 19  
 glPointSize, 14  
 glPolygonStipple, 17  
 glPolygonMode, 13  
 glPopAttrib, 73  
 glPopMatrix, 36  
 glPushAttrib, 73  
 glPushMatrix, 36  
 glRasterPos, 18  
 glReadBuffer, 70  
 glReadPixels, 72, 73  
 glRect, 13  
 glRotate, 36  
 glScale, 36  
 glScissor, 21, 22  
 glStencilFunc, 67  
 glStencilOp, 68  
 glTexCoord, 53  
 glTexCoordPointer, 25  
 glTexEnv, 52  
 glTexGen, 53  
 glTexImage2D, 47  
 glTexParameter, 49  
 glTranslate, 36  
 glVertex, 7  
 glVertexPointer, 25

glViewport, 39  
 wglUseFontBitmaps, 31  
     Команды GLU  
 gluBuild2DMipmaps, 49  
 gluCylinder, 24  
 gluDeleteQuadric, 23  
 gluDisk, 24  
 gluLookAt, 39  
 gluNewQuadric, 23  
 gluOrtho2D, 41  
 gluPartialDisk, 24  
 gluPerspective, 42  
 gluScaleImage, 46  
 gluSphere, 24  
 gluUnProject, 43  
 gluUnProject4, 43  
     Константы GL  
 GL\_BYTE, 46  
 GL\_FLOAT, 46  
 GL\_INT, 46  
 GL\_UNSIGNED\_BYTE, 46  
 GL\_UNSIGNED\_INT, 46  
 GL\_UNSIGNED\_SHORT, 46  
 GL\_ACCUM, 71  
 GL\_ACCUM\_BUFFER\_BIT, 7, 73  
 GL\_ADD, 71  
 GL\_ALL\_ATTRIB\_BITS, 73  
 GL\_ALPHA, 46  
 GL\_ALWAYS, 64, 67, 72  
 GL\_AMBIENT, 55, 58  
 GL\_AMBIENT\_AND\_DIFFUSE, 55  
 GL\_AND, 23  
 GL\_AND\_INVERTED, 23  
 GL\_AND\_REVERSE, 23  
 GL\_BACK, 13, 16  
 GL\_BITMAP, 46  
 GL\_BLUE, 46  
 GL\_CCW, 16  
 GL\_CLAMP, 50  
 GL\_CLEAR, 23  
 GL\_CLIP\_PLANEi, 21  
 GL\_COLOR\_ARRAY, 19, 25  
 GL\_COLOR\_BUFFER\_BIT, 7, 73  
 GL\_COLOR\_INDEX, 46  
 GL\_COMPILE, 28  
 GL\_COMPILE\_AND\_EXECUTE, 28  
 GL\_CONSTANT\_ATTENUATION,  
 57, 58  
 GL\_COPY, 23  
 GL\_COPY\_INVERTED, 23  
 GL\_CULL\_FACE, 16  
 GL\_CURRENT\_BIT, 73  
 GL\_CW, 16  
 GL DECR, 68  
 GL\_DEPTH\_BUFFER\_BIT, 7, 74

GL\_DEPTH\_TEST, 71  
 GL\_DIFFUSE, 55, 58  
 GL\_DONT\_CARE, 75  
 GL\_DST\_ALPHA, 65  
 GL\_DST\_COLOR, 65  
 GL\_EMISSION, 55  
 GL\_ENABLE\_BIT, 74  
 GL\_EQUAL, 64, 67, 72  
 GL\_EQUIV, 23  
 GL\_EVAL\_BIT, 74  
 GL\_EXP, 62  
 GL\_EXP2, 62  
 GL\_EYE\_LINEAR, 54  
 GL\_FASTEST, 75  
 GL\_FILL, 14  
 GL\_FOG\_BIT, 74  
 GL\_FOG\_COLOR, 63  
 GL\_FOG\_DENSITY, 62  
 GL\_FOG\_END, 62, 63  
 GL\_FOG\_HINT, 75  
 GL\_FOG\_MODE, 62  
 GL\_FOG\_START, 62  
 GL\_FRONT, 13, 16  
 GL\_FRONT\_BACK, 16  
 GL\_FRONT\_AND\_BACK, 13  
 GL\_GEQUAL, 64, 67, 72  
 GL\_GREATER, 64, 67, 72  
 GL\_GREEN, 46  
 GL\_HINT\_BIT, 74  
 GL\_INCR, 68  
 GL\_INVERT, 23, 68  
 GL\_KEEP, 68  
 GL\_LEQUAL, 64, 67, 72  
 GL\_LESS, 64, 67, 72  
 GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, 61  
 GL\_LIGHT\_MODEL\_TWO\_SIDE, 61  
 GL\_LIGHTING\_BIT, 74  
 GL\_LINE, 14  
 GL\_LINE\_BIT, 74  
 GL\_LINE\_LOOP, 10  
 GL\_LINE\_SMOOTH, 15  
 GL\_LINE\_SMOOTH\_HINT, 75  
 GL\_LINE\_STIPPLE, 15  
 GL\_LINE\_STRIP, 10  
 GL\_LINEAR, 50, 51, 62  
 GL\_LINEAR\_ATTENUATION, 57, 58  
 GL\_LINEAR\_MIPMAP\_LINEAR, 51  
 GL\_LINEAR\_MIPMAP\_NEAREST, 51  
 GL\_LINES, 10  
 GL\_LIST\_BIT, 74  
 GL\_LOAD, 71  
 GL\_MODELVIEW, 35  
 GL\_MODELVIEW\_MATRIX, 44  
 GL\_MODULATE, 52  
 GL\_MULT, 71

GL\_NAND, 23  
 GL\_NEAREST, 50, 51  
 GL\_NEAREST\_MIPMAP\_LINEAR, 51  
 GL\_NEAREST\_MIPMAP\_NEAREST, 51  
 GL\_NEVER, 64, 67, 72  
 GL\_NICETESR, 75  
 GL\_NOOP, 23  
 GL\_NOR, 23  
 GL\_NORMAL\_ARRAY, 19, 25  
 GL\_NORMALIZE, 14  
 GL\_NOTEQUAL, 64, 67, 72  
 GL\_OBJECT\_LINEAR, 54  
 GL\_ONE, 64  
 GL\_ONE\_MINUS\_DST\_ALPHA, 65  
 GL\_ONE\_MINUS\_DST\_COLOR, 65  
 GL\_ONE\_MINUS\_SRC\_ALPHA, 65  
 GL\_ONE\_MINUS\_SRC\_COLOR, 65  
 GL\_OR, 23  
 GL\_OR\_INVERTED, 23  
 GL\_OR\_REVERSE, 23  
 GL\_PERSPECTIVE\_CORRECTION\_HINT, 75  
 GL\_PIXEL\_MODE\_BIT, 74  
 GL\_POINT, 14  
 GL\_POINT\_BIT, 74  
 GL\_POINT\_SMOOTH, 14  
 GL\_POINT\_SMOOTH\_HINT, 75  
 GL\_POINTS, 9  
 GL\_POLYGON, 12  
 GL\_POLYGON\_BIT, 74  
 GL\_POLYGON\_SMOOTH\_HINT, 75  
 GL\_POLYGON\_STIPPLE\_BIT, 74  
 GL\_POSITION, 59  
 GL\_PROJECTION, 35  
 GL\_PROJECTION\_MATRIX, 44  
 GL\_QUAD\_STRIP, 12  
 GL\_QUADRATIC\_ATTENUATION, 57  
 GL\_QUADS, 12  
 GL\_RED, 46  
 GL\_REPEAT, 50  
 GL\_REPLACE, 52, 68  
 GL\_RETURN, 71  
 GL\_RGB, 46  
 GL\_RGBA, 46  
 GL\_SCISSOR\_BIT, 74  
 GL\_SCISSOR\_TEST, 21  
 GL\_SET, 23  
 GL\_SHININESS, 55  
 GL\_SHORT, 46  
 GL\_SMOOTH, 9  
 GL\_SPECULAR, 55, 58  
 GL\_SPHERE\_MAP, 54  
 GL\_SPOT\_CUTOFF, 57  
 GL\_SPOT\_DIRECTION, 59  
 GL\_SPOT\_EXPONENT, 57  
 GL\_SRC\_ALPHA, 65

GL\_SRC\_ALPHA\_SATURATE, 65  
GL\_SRC\_COLOR, 64  
GL\_STENCIL\_BUFFER\_BIT, 67  
GL\_STENCIL\_BUFFER\_BIT, 7, 74  
GL\_STENCIL\_TEST, 67  
GL\_TEXTURE, 35  
GL\_TEXTURE\_2D, 47  
GL\_TEXTURE\_BIT, 74  
GL\_TEXTURE\_COORD\_ARRAY, 19, 25  
GL\_TEXTURE\_ENV, 52  
GL\_TEXTURE\_ENV\_MODE, 52  
GL\_TEXTURE\_GEN\_MODE, 53  
GL\_TEXTURE\_MAG\_FILTER, 50  
GL\_TEXTURE\_MIN\_FILTER, 50  
GL\_TEXTURE\_WRAP\_S, 50  
GL\_TEXTURE\_WRAP\_T, 50  
GL\_TRANSFORM\_BIT, 74  
GL\_TRIANGLE\_FAN, 11  
GL\_TRIANGLE\_STRIP, 11  
GL\_TRIANGLES, 11  
GL\_VERTEX\_ARRAY, 19, 25  
GL\_VIEWPORTM\_BIT, 74  
GL\_VIEWPORTR, 44  
GL\_XOR, 23  
GL\_ZERO, 64, 68  
    Функции GDI  
CreateFont, 31





