

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ**

**МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ
РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)**

Факультет: Вычислительных машин и систем

КОМПЬЮТЕРНАЯ ГРАФИКА

**Методические указания по выполнению
лабораторной работы №2**

Тема: "Введение в графическую библиотеку OpenGL"

Составитель: Чертков К.К.

Москва, 2008

ЦЕЛЬ РАБОТЫ: познакомиться с основами программирования на OpenGL.

ЗАДАЧА: создать трехмерную модель заданного объекта в 3DS Max и программу на OpenGL для управления им.

1. Подключение и инициализация библиотеки

Для работы с OpenGL, как и с любой другой библиотекой, необходимы программные модули, содержащие описание всех констант и функций. Модули, поставляемые с компиляторами Delphi 7, VC++ 2005 и C++ Builder 6, предназначены для работы с очень старой версией 1.1 OpenGL. Сейчас уже существует версия 2.1. Кроме того, есть большое количество так называемых «расширений», значительно увеличивающих возможности графических программ и позволяющих использовать самые последние возможности современных видеокарт. Расширения разрабатываются фирмами-производителями графических процессоров (ATI, NVIDIA и др.) и постепенно включаются в стандарт OpenGL, иногда в несколько измененном виде. В прилагаемых примерах программ на Паскале использована библиотека `dglOpenGL` (поддерживает версию 2.0), на C++ — GLEW (взята из NVIDIA OpenGL SDK 10, поддерживает версию 2.1), на C# — Tao Framework (поддерживает версию 2.1 и дополнительно содержит несколько полезных библиотек подпрограмм).

Инициализация и работа со всеми библиотеками практически одинакова, поэтому дальше изложение будет вестись применительно к `dglOpenGL`. Эта библиотека представляет собой один модуль `dglOpenGL.pas`, который подключается стандартным способом:

```
uses dglOpenGL;
```

Перед работой с библиотекой необходимо вызвать функцию `InitOpenGL`, которая устанавливает указатели на все доступные функции ядра OpenGL и его расширений. Затем должен быть создан и задействован контекст отображения OpenGL.

```
var
  pfd : TPixelFormatDescriptor; //описание формата пикселей
  pf : Integer;                //индекс формата пикселей
begin

  InitOpenGL; //инициализация dglOpenGL

  //получим контекст устройства для окна
  dc := GetDC(Handle);

  //создадим описание требуемого формата пиксела
```

```

FillChar(pfd, sizeof(pfd), 0);
pfd.nSize := sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion := 1;
pfd.dwFlags := PFD_DRAW_TO_WINDOW + PFD_DOUBLEBUFFER +
               PFD_SUPPORT_OPENGL;
pfd.cStencilBits := 8;
pfd.cDepthBits := 24;

//получим номер наиболее близкого к запрошенным параметрам
//стандартного формата
pf := ChoosePixelFormat(dc, @pfd);

//получим его описание
DescribePixelFormat(dc, pf, sizeof(PIXELFORMATDESCRIPTOR), pfd);

//установим его для контекста устройства окна
SetPixelFormat(dc, pf, @pfd);

//создадим контекст отображения для OpenGL
hrc := wglCreateContext(dc);

//задействуем его
wglMakeCurrent(dc, hrc);

//установим флажки для всех доступных расширений OpenGL
ReadImplementationProperties;

```

Контекст отображения содержит информацию об используемом формате пикселей, куда входят данные об общем количестве цветов и способе их кодирования, о количестве бит, отводимых на один элемент буфера глубины и буфера шаблона и т.п. информация. Любая видеокарта поддерживает большой список форматов пикселей и программа при запуске должна выбрать из них наиболее подходящий для себя. Для этого нужно, прежде всего, создать «контекст устройства» окна. Он описывает характеристики графических устройств и используется всеми функциями рисования GDI, а также неявно — классом *TCanvas* из библиотеки *VCL Delphi*. Затем структура типа *TPixelFormatDescriptor* заполняется требуемыми характеристиками формата пикселя и передается функции *ChoosePixelFormat*. Та возвращает идентификатор формата, наиболее близкого к запрошенному (возвращенное значение 0 означает ошибку). Получить описание всех его характеристик можно с помощью функции *DescribePixelFormat*.

Первый элемент *nSize* вышеуказанной структуры *TPixelFormatDescriptor* должен быть равен ее размеру (40 байтов), второй — *nVersion* — всегда равен 1. В примере выше присваиваются значения еще двум полям: *cStencilBits* (число битов на один элемент буфера шаблона) и *cDepthBits* (число битов на один элемент буфера глубины). Если их оставить

равными нулю, то драйвер видеокарты может «решить», что эти буферы не нужны, и не создаст их. Еще одно поле *dwFlags* определяет свойства буфера кадра. Константа *PFD_DRAW_TO_WINDOW* указывает, что вывод информации будет производиться в окно программы (возможен также вывод в изображение BMP в памяти, но при этом нельзя использовать двойную буферизацию). *PFD_DOUBLEBUFFER* указывает, что изображение будет сначала выводиться в задний буфер, а затем, полностью сформированное, копироваться на экран. *PFD_SUPPORT_OPENGL* означает, что формат должен поддерживаться библиотекой OpenGL.

Команда *SetPixelFormat* устанавливает формат пикселей для контекста устройства окна, команда *wglCreateContext* создает контекст отображения OpenGL, команда *wglMakeCurrent* задействует его для вызывающего потока. После этого все вызовы команд OpenGL будут относиться к окну программы, на которое ссылался созданный ею контекст устройства.

Функция *ReadImplementationProperties* определяет, какие расширения OpenGL доступны и устанавливает соответствующие флажки в значение *true*. Например, если будет установлен флажок *WGL_EXT_swap_control*, то это означает, что программа может управлять синхронизацией копирования заднего буфера в передний с обратным ходом луча ЭЛТ (функция VSync). Благодаря этому изменение изображения на экране выглядит более плавным.

```
if (WGL_EXT_swap_control) then wglSwapIntervalEXT(1);
```

ReadImplementationProperties анализирует список расширений, возвращаемый командой *glGetString(GL_EXTENSIONS)*. Он представляет собой простую строку текста, где названия расширений разделены пробелами. Описание существующих расширений можно найти на сайте www.opengl.org, а также на сайтах developer.nvidia.com и ati.amd.com/developer/.

2. Системы координат. Проекция. Область вывода

В графическом конвейере OpenGL координаты вершин проходят несколько этапов преобразования:

1. Преобразование из локальной системы координат объекта в мировую систему координат, а затем — в систему координат камеры.
2. Преобразование из системы координат камеры в левую систему координат, перспективное преобразование (если задано), преобразование к системе координат отсечения (при этом заданный видимый объем переходит в канонический видимый объем в виде куба).

3. Отсечение границами канонического видимого объема и нормализация однородных координат (преобразование в нормализованные координаты устройства). В данном случае нормализацию однородных координат принято также называть перспективным делением, так как в результате мы получаем координаты объектов с учетом перспективы.
4. Преобразование в оконную систему координат с учетом размеров и положения области просмотра в окне программы.

Преобразование в мировую систему координат и систему координат камеры

Это преобразование описывается так называемой матрицей «модель-вид». Она объединяет в себе преобразования, определяющие положение изображаемого объекта и камеры в мировой системе координат. Для того, чтобы ее задать, нужно с помощью команды *glMatrixMode* переключиться в режим изменения матрицы «модель-вид». Затем, например, с помощью команды *gluLookAt* определить положение камеры в пространстве. После этого полученную матрицу можно с помощью команды *glMultMatrixd* домножить на матрицу преобразования из локальной системы координат объекта в мировую систему координат:

```
//преобразование в мировую систему координат
//из локальной системы координат объекта
var tm : TTransform = (...);

.....

//перейдем в режим изменения матрицы «модель-вид»
glMatrixMode(GL_MODELVIEW);

//заменяем старую матрицу «модель-вид» единичной
glLoadIdentity();

//умножим матрицу «модель-вид» на матрицу преобразования
//из мировой системы координат в систему координат камеры
gluLookAt(0, 100, 0, 0, -20, -500, 0, 1, 0);

//домножим матрицу «модель-вид» на матрицу преобразования
//из локальной системы координат объекта в мировую
//систему координат
glMultMatrixd(@tm[0,0]);
```

Параметры (*eyex, eyey, eyez*) команды *gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz : double)* задают положение точки наблюдения *Eye* (начала системы координат камеры). Параметры (*centerx, centery, centerz*) определяют координаты центральной точки сцены

Center, на которую она должна быть направлена. Параметры (*upx*, *upy*, *upz*) указывают ее ориентацию относительно оси *Eye – Center* и являются координатами вектора \vec{Up} , направленного вверх с точки зрения камеры. Этот вектор не обязательно должен быть перпендикулярен к оси *Eye – Center*, но и не может быть параллелен ей. Система координат, определяемая данной командой, — привычная правая (рис. 1). Все координаты, передаваемые функции *gluLookAt*, задаются относительно мировой системы координат.

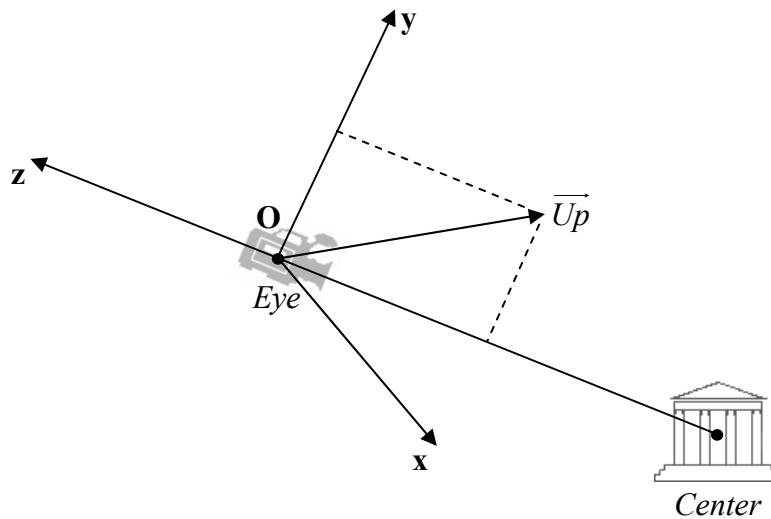


Рис. 1. Определение системы координат камеры командой *gluLookAt*

Управление камерой

Камера в компьютерной графике — это система координат, точка наблюдения, плоскость проекций и видимый объем. Поэтому задача перемещения или поворота камеры состоит в перемещении начала или изменении ориентации осей ее системы координат. Начальное положение камеры в пространстве обычно задается командой *gluLookAt*, описанной выше. Она умножает текущую матрицу «модель-вид» $[MV]$ на матрицу $[C]$ преобразования из мировой системы координат в систему координат камеры: $[MV] \leftarrow [MV] \cdot [C]$. Перед вызовом *gluLookAt* $[MV]$, как правило, устанавливается равной единичной матрице и поэтому затем становится равной $[C]$. Матрицу $[C]$ удобно хранить в отдельном массиве и загружать перед каждым рисованием сцены. В примере ниже камера устанавливается в исходное положение с помощью команды *gluLookAt*, после чего созданная ею матрица копируется в массив *C*.

```
//матрица преобразования из мировой системы
//координат в систему координат камеры
```

```

var C : TTransform;

.....

//перейдем в режим изменения матрицы "модель-вид"
glMatrixMode(GL_MODELVIEW);

//заменим старую матрицу единичной
glLoadIdentity;

//установим камеру в начальное положение
gluLookAt(0, 100, 0, 0, -20, -500, 0, 1, 0);

//сохраним матрицу, описывающую положение
//камеры в пространстве, в массиве C
glGetDoublev(GL_MODELVIEW_MATRIX, @C[0,0]);

```

Повороты и перемещения камеры могут совершаться как относительно осей ее собственной системы координат, так и относительно осей мировой системы координат. В первом случае матрица $[C]$ должна быть умножена на матрицу, обратную к матрице $[T]$ поворота или переноса слева: $[C] \leftarrow [T]^{-1} \cdot [C]$, во втором случае — справа: $[C] \leftarrow [C] \cdot [T]^{-1}$. В следующем примере камера поворачивается на $+5^\circ$ вокруг локальной оси y . Обратите внимание, что для этого используется команда поворота в обратную сторону на -5° .

```

//матрица преобразования из мировой системы
//координат в систему координат камеры
var C : TTransform;

.....

//перейдем в режим изменения матрицы "модель-вид"
glMatrixMode(GL_MODELVIEW);

//повернем камеру на +5 градусов, умножив матрицу [C]
//слева на матрицу поворота на -5 градусов
glLoadIdentity;
glRotated(-5, 0, 1, 0); //поворот вокруг оси Y на -5 градусов
glMultMatrixd(@C[0,0]); //умножим справа на матрицу [C]

//сохраним матрицу, описывающую положение
//повернутой камеры в пространстве, в массиве C
glGetDoublev(GL_MODELVIEW_MATRIX, @C[0,0]);

```

Проекции

В OpenGL предусмотрены команды для построения перспективной (*glFrustum* и *gluPerspective*) и ортогографической (*glOrtho* и *gluOrtho2D*) проекций. Преобразование координат, которое они задают, состоит из трех отдельных преобразований:

1. Переход от правой системы координат камеры, показанной на рис. 1, к левой системе координат. При этом направление оси z меняется на противоположное так, что у всех объектов, расположенных перед камерой, координата z становится положительной.
2. Перспективное преобразование (для команд *glFrustum* и *gluPerspective*), которое дает изображение сцены в перспективе. При этом видимый объем, заданный усеченной пирамидой (рис. 2), переходит в параллелепипед.
3. Преобразование видимого объема в канонический видимый объем (рис. 3) в виде куба с центром в начале координат и длиной ребра, равной двум. Для этого применяются преобразования параллельного переноса и масштабирования. Полученные в результате координаты называются координатами отсечения, так как за этим следует отсечение целых объектов или их отдельных частей, выходящих за границы видимого объема.

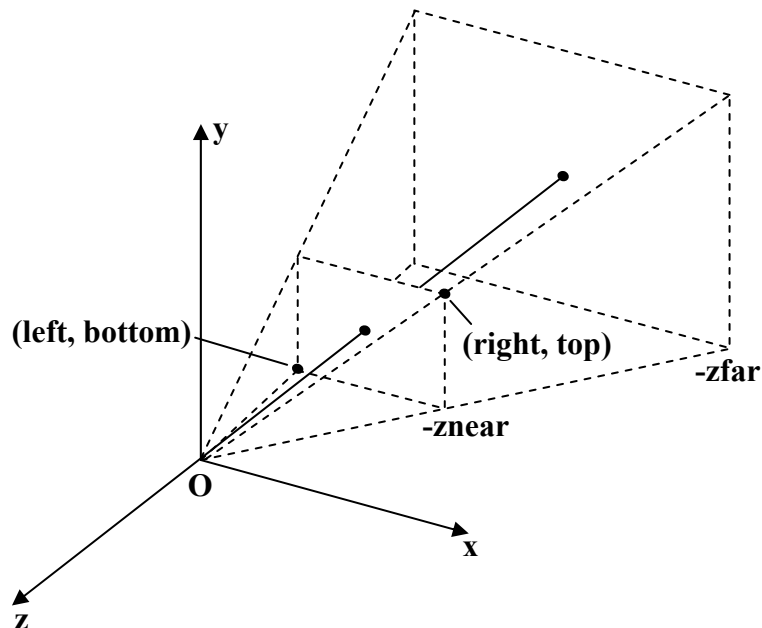


Рис. 2. Видимый объем, определяемый командами *glFrustum* и *gluPerspective*

Значение параметров команды *glFrustum*(*left*, *right*, *bottom*, *top*, *znear*, *zfar* : *double*) ясно из рис. 2. Параметры *znear* и *zfar* определяют аппликату соответственно ближней и дальней плоскостей отсечения видимого объема в системе координат камеры с измененным на обратное направлением оси *z*. При перспективном преобразовании видимый объем всегда должен располагаться перед камерой, поэтому эти параметры всегда должны быть положительными. Команда *gluPerspective*(*fovy*, *aspect*, *znear*, *zfar* : *double*) более удобна, так как без дополнительных расчетов позволяет учесть соотношение высоты и ширины области просмотра. Если отношение ширины и высоты видимого объема не будет совпадать с отношением ширины и высоты области просмотра на экране, то изображение будет искаженным: сжатым или растянутым. Последние два параметра данной команды полностью аналогичны таким же параметрам команды *glFrustum*. Параметр *fovy* задает в градусах угол обзора по вертикали, параметр *aspect* — отношение ширины видимого объема к его высоте.

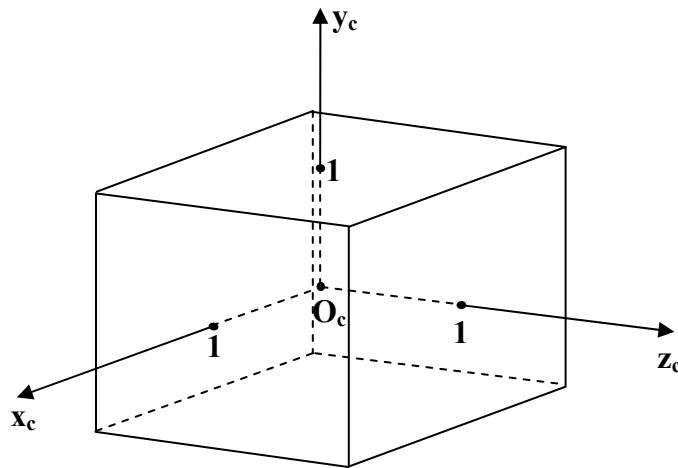


Рис. 3. Канонический видимый объем и система координат отсечения

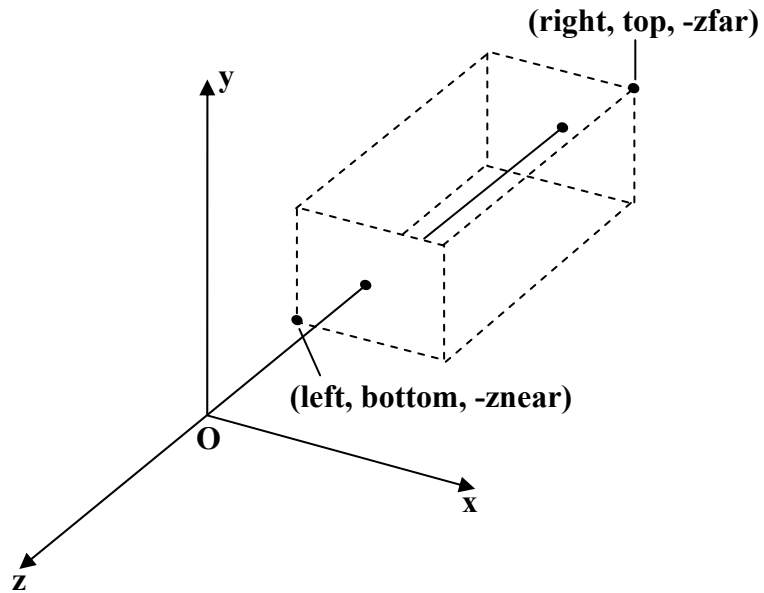


Рис. 4. Видимый объем, определяемый командами *glOrtho* и *gluOrtho2D*

Видимый объем, определяемый командой *glOrtho(left, right, bottom, top, znear, zfar : double)*, представляет собой параллелепипед (рис. 4). Как и в случае команды *glFrustum*, положительные значения *znear* и *zfar* соответствуют пространству перед камерой. При ортографической проекции видимый объем можно расположить и за камерой. Для этого два указанных параметра нужно задать отрицательными. Команду *gluOrtho2D(left, right, bottom, top: double)* удобно использовать при изображении двумерных сцен. Ее вызов аналогичен вызову *glOrtho* с параметрами *znear = -1, zfar = 1*.

Описанные в данном разделе преобразования задаются матрицей проекции. Для того, чтобы ее определить, нужно, как и при изменении матрицы «модель-вид», сначала переключиться в соответствующий режим:

```
//перейдем в режим изменения матрицы проекции
glMatrixMode(GL_PROJECTION);

//заменяем старую матрицу единичной
glLoadIdentity();

//домножим на матрицу перспективного преобразования
gluPerspective(25, ClientWidth / ClientHeight, 300, 700);
```

Если камера должна быть установлена очень близко к изображаемому объекту, то удобнее расположить точку наблюдения не в начале ее системы координат, как это делают команды *glFrustum* и *gluPerspective*, а на некотором удалении на положительной полуоси z . В этом случае необходимо воспользоваться следующим перспективным преобразованием:

$$P_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1/Z_s \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Кроме уменьшения размеров удаленных объектов по ширине и высоте, данное преобразование сжимает пространство, находящееся за плоскостью Oxy так, что оно переходит в пространство между плоскостями Oxy и $z = -Z_s$. В результате объем в виде усеченной пирамиды с бесконечно удаленной задней плоскостью, показанный на рис. 5 (а), перейдет в параллелепипед, показанный на рис. 5 (b). Поэтому для того, чтобы нарисовать объекты, находящиеся на любом, даже очень большом расстоянии от камеры, достаточно координату дальней плоскости отсечения видимого объема задать равной $-Z_s$:

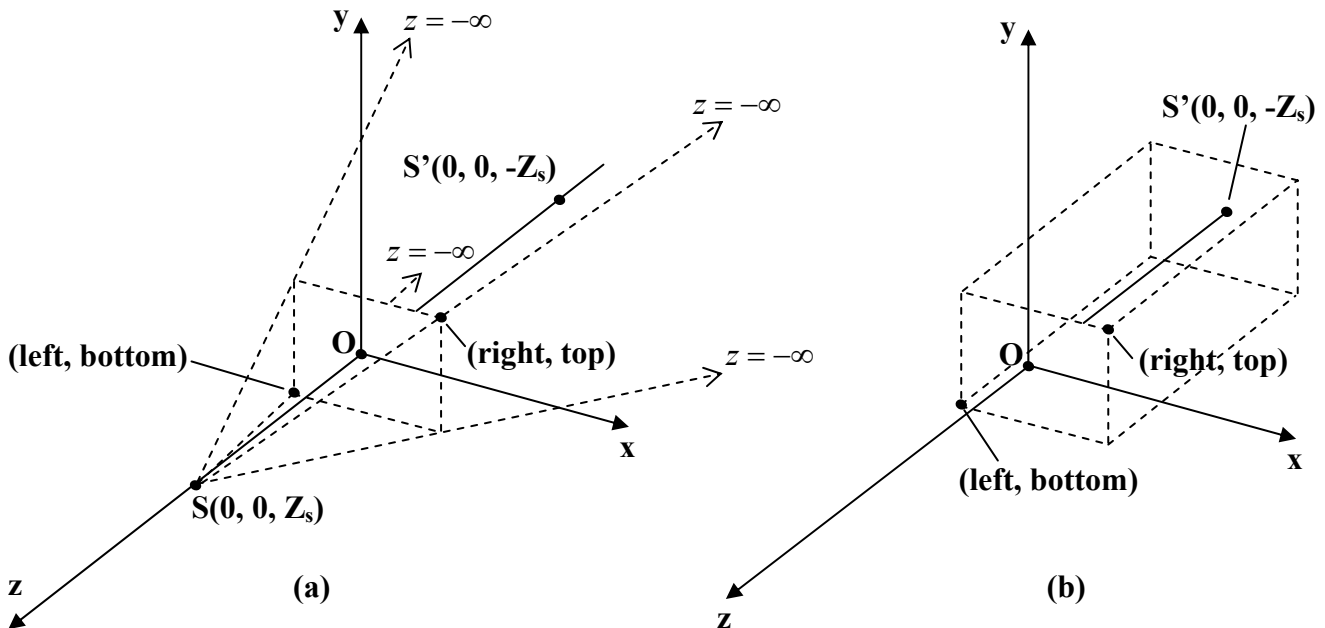


Рис. 5. Перспективное преобразование с центром, отличным от начала координат

```
//координата Z точки наблюдения S
const Zs = 500;
```

```

//перспективное преобразование
var Pp : TTransform = (
    (1, 0, 0, 0),
    (0, 1, 0, 0),
    (0, 0, 1, -1/Zs),
    (0, 0, 0, 1));

.....

//перейдем в режим изменения матрицы проекции
glMatrixMode(GL_PROJECTION);

//заменяем старую матрицу единичной
glLoadIdentity();

//умножим её на матрицу преобразования в канонический
//видимый объем
glOrtho(-100, 100, -100, 100, 0, 500);

//умножим на матрицу перспективного преобразования
glMultMatrixd(@Pp[0,0]);

```

Оконная система координат и область просмотра

Оконная система координат OpenGL $O_w x_w y_w z_w$ показана на рис. 6. Это левая система координат и ее ось z_w направлена от наблюдателя вглубь экрана. Именно к этой системе координат в конечном итоге приводятся координаты всех объектов перед их изображением. Для этого «координаты отсечения» (x_c, y_c, z_c, w_c) вершин, оставшихся после отсечения границами канонического видимого объема, проходят еще два этапа преобразования. Сначала они нормализуются (выполняется перспективное деление), в результате чего получаются обычные

(неоднородные) координаты $(x_{nd}, y_{nd}, z_{nd}) = \left(\frac{x_c}{w_c}, \frac{y_c}{w_c}, \frac{z_c}{w_c} \right)$ (в OpenGL для их названия

используется термин «нормализованные координаты устройства»). Затем они подвергаются следующему аффинному преобразованию, зависящему от положения и размеров области просмотра (рис. 6):

$$x_w = (x_{nd} + 1) \cdot \frac{width}{2} + x, \quad y_w = (y_{nd} + 1) \cdot \frac{height}{2} + y, \quad z_w = \frac{zfar - znear}{2} \cdot z_{nd} + \frac{znear + zfar}{2}.$$

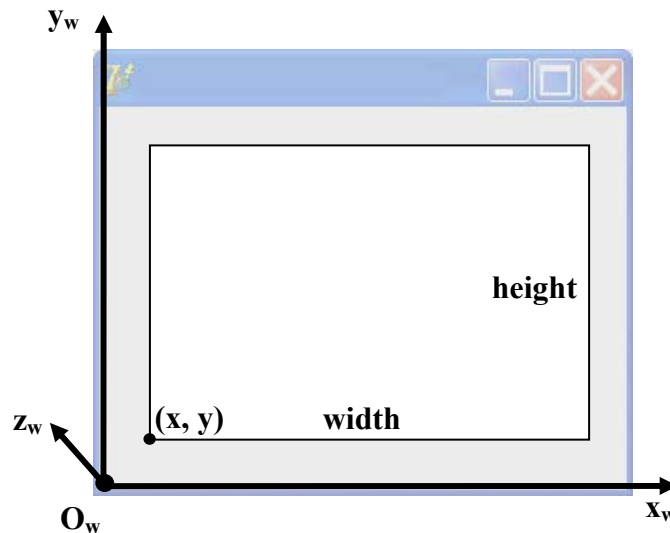


Рис. 6. Определение области просмотра в оконной системе координат

Область просмотра задается командой `glViewport(x, y, width, height : integer)`. Преобразование координаты z_{nd} зависит от двух параметров: z_{near} и z_{far} , которые можно определить с помощью команды `glDepthRange(znear, zfar : double)`. По умолчанию их значения равны соответственно 0 и 1, так что самые дальние вершины, лежащие на дальней плоскости отсечения $z_c = 1$ канонического видимого объема (рис. 3) будут иметь координату $z_w = 1$, а самые ближние, лежащие на ближней плоскости отсечения $z_c = -1$, будут иметь координату $z_w = 0$.

3. Освещение и материалы

В модели освещения OpenGL свет искусственно делится на три составляющих: рассеянную, диффузную, и зеркальную. Первая моделирует свет, отражающийся от окружающих предметов (стен, потолка и пр.) и задает минимальный уровень освещенности сцены I_a . Диффузная I_d и зеркальная I_s составляющие описывают вклад источника света в его соответственно диффузное и зеркальное отражение поверхностью объектов. Для описания свойств поверхности задаются коэффициенты отражения каждой из этих составляющих: K_a , K_d и K_s . Кроме того, при зеркальном отражении еще учитывается степень гладкости поверхности γ , которая может принимать значения от 0 до 128. Чем лучше поверхность отполирована, тем больше должно быть это значение. Ниже приведена основная формула, по

которой происходит расчет яркости каждой из компонент R , G , B отраженного света (здесь не учитывается ослабление света с расстоянием и некоторые другие параметры):

$$I = I_a \cdot K_a + I_d \cdot K_d \cdot \max(\cos(\theta), 0) + I_s \cdot K_s \cdot \max(\cos(\beta), 0)^{\gamma}. \quad (*)$$

В этой формуле предполагается, что длины всех векторов равны единице и косинусы углов β и θ находятся через скалярное произведение: $\cos(\theta) = (\vec{L}, \vec{N})$, $\cos(\beta) = (\vec{H}, \vec{N})$. Поэтому необходимо следить за тем, чтобы вектора нормалей, которые передаются графическому конвейеру, были всегда нормализованы. В противном случае освещение будет рассчитываться неправильно. Можно включить их автоматическую нормализацию командой `glEnable(GL_NORMALIZE)`, но это несколько замедлит расчеты. Тем не менее, если объект подвергается преобразованию масштабирования, сделать это необходимо.

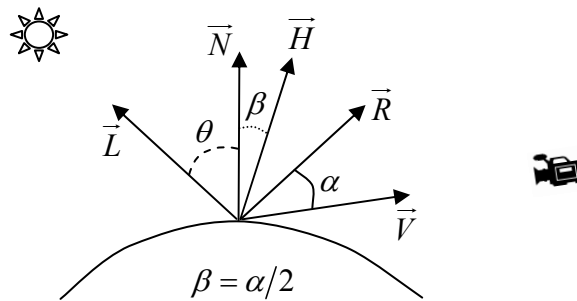


Рис. 7. Расчет отражения света в вершине (\vec{L} – вектор направления на источник света, \vec{N} – вектор нормали к поверхности, \vec{R} – вектор зеркального отражения света, \vec{V} – вектор направления на камеру, \vec{H} – биссектриса угла \widehat{LV})

Управление освещением

Для полигональных поверхностей расчет освещения происходит только в вершинах. Если включена закрашка Гуро со сглаживанием (команда `glShadeModel(GL_SMOOTH)`), то для определения цвета промежуточных точек граней используется билинейная интерполяция. Если включена однотонная закрашка (команда `glShadeModel(GL_FLAT)`), то для всех точек грани берется цвет одной из ее вершин.

Включение расчета освещения производится командой `glEnable(GL_LIGHTING)`, отключение — `glDisable(GL_LIGHTING)`. Если освещение отключено, то вершинам назначается цвет, заданный командами `glColor4f(red, green, blue, alpha : single)`, `glColor3f(red, green, blue : single)`, `glColor4fv(v : ^single)`, `glColor3fv(v : ^single)` и т.п. Для большинства команд OpenGL

существует несколько вариантов, отличающихся типом и количеством параметров, а также способом их передачи. Далее мы будем упоминать только о некоторых из них. Описание остальных можно найти в справочнике по OpenGL.

В OpenGL можно использовать одновременно до восьми источников света, именуемых константами `GL_LIGHT0` ... `GL_LIGHT7`. Чтобы включить, например, 1-й, нужно опять использовать команду `glEnable(GL_LIGHT0)`, а чтобы выключить — `glDisable(GL_LIGHT0)`.

Характеристики источника света задаются командой `glLightfv(light, pname: integer; param: ^single)` или ее аналогом. Как первый параметр указывается источник света, как второй — изменяемое свойство, как третий — присваиваемое ему значение. Константы `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR` соответствуют рассеянной, диффузной и зеркальной составляющей света от источника.

Чтобы задать положение источника света в пространстве, нужно использовать константу `GL_POSITION`. При этом можно указать координаты обычной точки, например, $(100, 0, 0, 1)$, а можно — бесконечно удаленной, например, $(1, 0, 0, 0)$. В первом случае источник света будет расположен на конечном расстоянии, и для каждой вершины вектор \vec{L} в уравнении (*) будет вычисляться отдельно. Во втором случае источник света будет расположен в бесконечности в положительном направлении оси абсцисс, и лучи света, падающие от него, будут параллельны друг другу. Координаты вектора \vec{L} для всех вершин будут приняты равными координатам источника света после нормализации. Первый способ, хотя и требует больше вычислений, но дает лучший результат. Нужно отметить, что передаваемые команде `glLightfv` координаты источников света подвергаются текущему преобразованию «модель-вид». Это позволяет изменять их положение в пространстве, используя для расчетов конвейер OpenGL.

С помощью команды `glLightModeli` можно управлять положением точки наблюдения в системе координат камеры, которое влияет на вычисление вектора \vec{V} . Вызов `glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE)` указывает, что точка наблюдения будет находиться в начале системы координат камеры. При этом вектор \vec{V} будет вычисляться для каждой вершины отдельно. Если последний параметр указать равным `GL_FALSE`, то точка наблюдения будет считаться расположенной на очень большом расстоянии в направлении оси Z (говоря языком математики, — в бесконечности), и вектор \vec{V} будет всегда иметь координаты $(0, 0, 1, 0)$. Первый режим дает более правдоподобное изображение, но требует больше вычислений.

Ниже приводится пример настройки параметров одного источника света.

```
var
  //положение источника света в пространстве
  light_position : array[0..3] of single = (0, 100, 400, 1.0);
  //рассеянная составляющая
  light_ambient : array[0..3] of single = (0.1, 0.1, 0.1, 1.0);
  //диффузная составляющая
  light_diffuse : array[0..3] of single = (0.6, 0.6, 0.6, 1.0);
  //зеркальная составляющая
  light_specular : array[0..3] of single = (0.5, 0.5, 0.5, 1.0);

  .....

  //включим расчет освещения
  glEnable(GL_LIGHTING);

  //включим источник 0
  glEnable(GL_LIGHT0);

  //определим его параметры
  glLightfv(GL_LIGHT0, GL_AMBIENT, @light_ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @light_diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, @light_specular);
  glLightfv(GL_LIGHT0, GL_POSITION, @light_position);
```

В OpenGL можно отдельно задать уровень рассеянного освещения сцены, независимый от наличия источников света. Для этого используется команда *glLightModelfv(pname : integer; params : ^single)*. Как первый параметр ей нужно передать константу *GL_LIGHT_MODEL_AMBIENT*, как второй — указатель на вектор (*r, g, b, a*) компонентов рассеянного света:

```
//рассеянное освещение сцены
sa : array[0..3] of single = (0.2, 0.2, 0.2, 1.0);
.....

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @sa);
```

Управление материалами

Для настройки свойств материала используются команды *glMaterialf(face, pname : integer; param : single)*, *glMaterialfv(face, pname : integer; param : ^single)* и т.п. Первый параметр может принимать значения *GL_FRONT*, *GL_BACK* и *GL_FRONT_AND_BACK*, указывая, что

описываются свойства материала соответственно для лицевых, обратных или одновременно для тех и других граней. Второй параметр указывает изменяемое свойство и может принимать значения *GL_AMBIENT* (коэффициенты отражения рассеянного света), *GL_DIFFUSE* (коэффициенты диффузного отражения света), *GL_SPECULAR* (коэффициенты зеркального отражения света), *GL_EMISSION* (яркость свечения материала), *GL_SHININESS* (степень γ в уравнении (*)) и *GL_AMBIENT_AND_DIFFUSE* (аналогично двум вызовам с константами *GL_AMBIENT* и *GL_DIFFUSE*).

По умолчанию диффузное и рассеянное отражение света рассчитывается только для лицевых граней. Обратные стороны граней освещаются лишь рассеянным светом. Чтобы они освещались полностью, нужно использовать команду *glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE)*. Если ее вызвать, указав как последний параметр *GL_FALSE*, то система вернется в исходный режим. Ниже приводится пример настройки свойств материала.

```
var
  //отражение рассеянного света
  material_ambient : array[0..3] of single = (0.5, 0.5, 0.0, 1.0);
  //диффузное отражение света
  material_diffuse : array[0..3] of single = (0.5, 0.5, 0.0, 1.0);
  //зеркальное отражение света
  material_specular : array[0..3] of single = (0.9, 0.9, 0.9, 1.0);
  //свечение
  material_selfillum : array[0..3] of single = (0.0, 0.0, 0.0, 1.0);

  .....

  //установим свойства материала
  glMaterialfv(GL_FRONT, GL_AMBIENT, @material_ambient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, @material_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, @material_specular);
  glMaterialfv(GL_FRONT, GL_EMISSION, @material_selfillum);
  glMaterialf(GL_FRONT, GL_SHININESS, 20); //степень гамма
```

4. Буферы OpenGL

В OpenGL программисту доступны несколько видов буферов, основными из которых являются цветовой буфер и буфер глубины. В первом формируется и хранится изображение, второй используется алгоритмом удаления невидимых поверхностей и хранит расстояние точек объектов до наблюдателя (координату z). Он также называется z-буфером. Перед рисованием каждого кадра эти буферы нужно очищать:

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

Первая команда задает цвет, которым будет заполняться цветовой буфер, вторая производит очистку обоих буферов. Для заполнения буфера глубины по умолчанию используется значение 1 (наибольшее расстояние до плоскости проекций в оконной системе координат).

Если при инициализации библиотеки параметр *dwFlags* структуры *TPixelFormatDescriptor* содержал флажок *PFD_DOUBLEBUFFER*, то изображение, выводимое программой, будет помещаться в задний буфер. Затем его нужно скопировать в передний буфер командой *SwapBuffers(hdc : HDC)* и тогда оно станет видимым. В качестве параметра этой функции передается идентификатор контекста устройства окна программы (про контекст устройства см. в разделе «Подключение и инициализация библиотеки»).

Удаление невидимых поверхностей

OpenGL предоставляет два метода для удаления невидимых поверхностей: алгоритм с z-буфером и отсечение граней. Первый включается командой *glEnable(GL_DEPTH_TEST)* и выключается командой *glDisable(GL_DEPTH_TEST)*. Второй — соответственно командами *glEnable(GL_CULL_FACE)* и *glDisable(GL_CULL_FACE)*. По умолчанию отсекаются нелицевые грани, то есть грани, вершины которых с точки зрения наблюдателя расположены в порядке по часовой стрелке. Чтобы увидеть внутренность объектов, можно включить отсечение лицевых граней. Это делается с помощью команды *glCullFace(mode : integer)*. Ее параметр *mode* может принимать значения *GL_FRONT* (отсечение лицевых граней), *GL_BACK* (отсечение нелицевых граней), *GL_FRONT_AND_BACK* (отсечение всех граней). С помощью команды *glFrontFace(mode : integer)* можно также указать, какой порядок вершин будет соответствовать лицевым граням. Константа *GL_CCW* задает порядок против часовой стрелки, константа *GL_CW* — по часовой.

5. Работа с матрицами. Базовые преобразования координат

В OpenGL используется три вида матриц: «модель-вид», проекции и преобразования текстурных координат. Матрица «модель-вид» описывает положение объектов и камеры в пространстве (в мировой системе координат) и обычно содержит комбинацию преобразований

из локальной системы координат объекта в мировую и из мировой в систему координат камеры. Матрица проекции определяет проективное преобразование и видимый объем, то есть ту часть пространства, которая будет изображена на экране. Матрица преобразования текстурных координат применяется для создания различных эффектов от простого поворота или масштабирования текстуры до построения теней и зеркальных отражений.

Для того, чтобы изменить заданную матрицу, нужно переключиться в соответствующий режим. Для этого используется команда *glMatrixMode(mode : integer)*, которой в качестве параметра передается одна из констант *GL_MODELVIEW*, *GL_PROJECTION* или *GL_TEXTURE*. После этого все команды умножения и загрузки матриц будут действовать на выбранную матрицу.

Базовые преобразования координат

Для задания базовых преобразований координат в OpenGL предусмотрены три команды: *glTranslated(dx, dy, dz : double)*, *glRotated(angle, x, y, z : double)* и *glScaled(kx, ky, kz : double)*, а также их модификации *glTranslatef*, *glRotatef*, *glScalef* с параметрами типа *single*.

Команда *glTranslated(dx, dy, dz : double)* умножает текущую матрицу $[M]$ справа на матрицу параллельного переноса: $[M] \leftarrow [M] \cdot [T]$,

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}^T.$$

Команда *glRotated(angle, x, y, z : double)* умножает текущую матрицу $[M]$ на матрицу поворота на угол *angle* вокруг оси с направляющим вектором $\vec{l} = \{x, y, z\}$. Например, в результате вызова *glRotated(30, 0, 0, 1)* матрица $[M]$ будет умножена на матрицу поворота вокруг оси *z* на 30°: $[M] \leftarrow [M] \cdot [R_z]$,

$$[R_z] = \begin{bmatrix} \cos(30^\circ) & \sin(30^\circ) & 0 & 0 \\ -\sin(30^\circ) & \cos(30^\circ) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^T$$

Команда *glScaled(kx, ky, kz : double)* используется для построения преобразований масштабирования и отражения. Например, для того, чтобы добавить к имеющемуся

преобразованию $[M]$ преобразование отражения относительно плоскости Oxy , нужно использовать вызов `glScaled(1, 1, -1)`. В результате получим следующее произведение матриц:

$$[M] \leftarrow [M] \cdot [M_{Oxy}],$$

$$[M_{Oxy}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Заменить текущую матрицу $[M]$ единичной можно, вызвав команду `glLoadIdentity`:

$$[M] \leftarrow [E],$$

$$[E] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Как пример приведем команды, задающие поворот на 45° вокруг оси $\begin{cases} x = -200 \\ y = -400 \end{cases}$:

```
//перейдем в режим изменения матрицы "модель-вид"
glMatrixMode(GL_MODELVIEW);

//заменяем старую матрицу единичной
glLoadIdentity;

//возврат к исходной системе координат
glTranslated(-200, -400, 0);

//поворот вокруг оси z на 45 градусов
glRotated(45, 0, 0, 1);

//перенос системы координат для
//совмещения оси z с осью вращения
glTranslated(+200, +400, 0);
```

Обратите внимание на обратный порядок преобразований. Это связано с тем, что в OpenGL используются транспонированные матрицы. К примеру, если преобразование точки (x, y, z, w) в точку (x', y', z', w') привычно задается так:

$[x' \ y' \ z' \ w'] = [x \ y \ z \ w] \cdot [T_1] \cdot [T_2] \cdot [T_3]$, то в OpenGL оно происходит так:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = [T_3]^T \cdot [T_2]^T \cdot [T_1]^T \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}.$$

Загрузка и извлечение матриц

OpenGL позволяет использовать любые матрицы, определенные программистом. Для этого применяются команды `glMultMatrixd(m : ^double)` и `glLoadMatrixd(m : ^double)`. Первая умножает, а вторая заменяет текущую матрицу преобразования переданной в качестве параметра. Загружаемые матрицы OpenGL транспонирует (считая, что они расположены в памяти столбец за столбцом), поэтому их нужно задавать в обычном виде. Приведенный ниже пример показывает, как можно использовать собственную матрицу $[P]$ параллельного переноса. Она объединяется с текущей матрицей $[M]$ следующим образом: $[M] \leftarrow [M] \cdot [P]^T$.

```
var P : TTransform = (
  ( 1, 0, 0, 0),
  ( 0, 1, 0, 0),
  ( 0, 0, 1, 0),
  (100, 200, 300, 1));

.....

glMultMatrixd(@P[0,0]);
```

Получить текущую матрицу «модель-вид», проекции или преобразования текстурных координат можно, вызвав команду `glGetDoublev(pname : integer; params : ^double)`. Как первый параметр нужно указать константу `GL_MODELVIEW_MATRIX`, `GL_PROJECTION_MATRIX` или `GL_TEXTURE_MATRIX`, как второй — указатель на массив, куда будет скопирована матрица. Приведенный ниже пример показывает, как с помощью OpenGL можно получить матрицу R поворота на 60° вокруг оси $\vec{l} = \{1, 1, 1\}$.

```
var R : TTransform;
```

```

.....

//перейдем в режим изменения матрицы "модель-вид"
glMatrixMode(GL_MODELVIEW);

//заменяем старую матрицу единичной
glLoadIdentity();

//умножим на матрицу поворота
glRotated(60, 1, 1, 1);

//скопируем матрицу поворота в массив R
glGetDoublev(GL_MODELVIEW_MATRIX, @R[0, 0]);

```

Стек матриц

В OpenGL матрицы образуют стек, на самом верху которого находится матрица текущего преобразования $[M]$. Для работы со стеком используются команды *glPushMatrix()* и *glPopMatrix()*. Первая создает копию $[M]$ и добавляет ее в стек как новую матрицу. Вторая удаляет верхнюю матрицу из стека и текущей становится расположенная под ней. Следующий пример демонстрирует использование стека для сохранения матрицы $[tm]$, на основе которой строятся два разных преобразования координат.

```

//матрица преобразования из локальной системы координат
//главного объекта в мировую систему координат
tm : TTransform;

//матрицы преобразования из локальной системы координат
//зависимых объектов в локальную систему координат
//главного объекта
rtm1, rtm2 : TTransform;

.....

//перейдем в режим изменения матрицы "модель-вид"
glMatrixMode(GL_MODELVIEW);
//загрузим матрицу tm
glLoadMatrixd(@tm[0,0]);

//нарисуем главный объект
.....

//сохраним матрицу tm в стеке
glPushMatrix();
//умножим матрицу tm на матрицу rtm1
glMultMatrixd(@rtm1[0,0]);

//нарисуем зависимый объект 1
.....

```

```

//восстановим матрицу tm
glPopMatrix;

//сохраним матрицу tm в стеке
glPushMatrix;
//домножим матрицу tm на матрицу rtm2
glMultMatrixd(@rtm1[0,0]);

//нарисуем зависимый объект 2
.....

//восстановим матрицу tm
glPopMatrix;

```

6. Рисование графических примитивов

Определение вершин и нормалей

В OpenGL любой геометрический объект задается множеством вершин. Для их определения предназначены команды *glVertex*(...)*: *glVertex2d(x, y : double)*, *glVertex3d(x, y, z : double)*, *glVertex4d(x, y, z, w : double)*, *glVertex2dv(v : ^ double)*, *glVertex3dv(v : ^ double)*, *glVertex4dv(v : ^ double)* и т.п. Каждая вершина в OpenGL описывается четырьмя однородными координатами (x, y, z, w). Если координаты z и w не были заданы, то им присваиваются соответственно значения 0 и 1. Для определения нормалей предназначены команды *glNormal*(...)*: *glNormal3d(nx, ny, nz : double)*, *glNormal3dv(nx, ny, nz : double)* и т.п. Команда *glVertex*(...)* завершает определение вершин. То есть, команды, задающие цвет вершины, координаты нормали к ней и пр. должны вызываться до *glVertex*(...)*.

В следующем примере определяются две вершины с однородными координатами (10, 10, 10, 1), (5, 5, 0, 1) и вектора нормалей к ним с координатами (1, 0, 0) и (0, 0, 1).

```

var
  V1 : TVertex = (10, 10, 10, 1);
  N1 : TVector3 = (1, 0, 0);

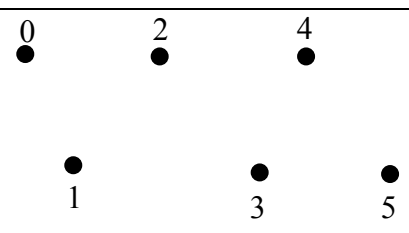
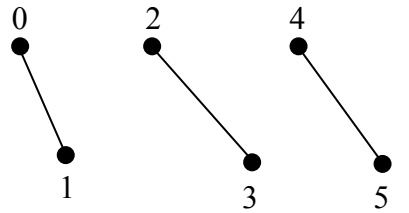
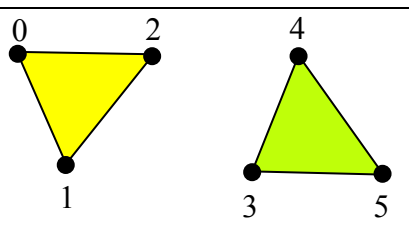
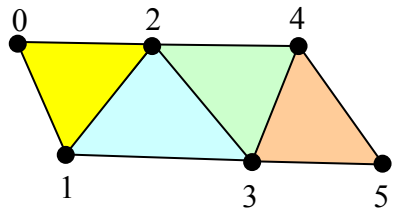
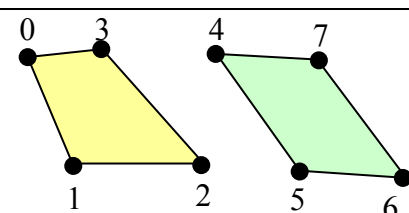
  glNormal3dv(@N1);
  glVertex4dv(@V1);

  glNormal3d(0, 0, 1);
  glVertex2d(5, 5);

```

Графические примитивы

Для того, чтобы указать OpenGL, как обрабатывать вершины, вызовы *glVertex*()*, *glNormal*()* и других подобных команд должны быть заключены в операторные скобки *glBegin(mode : integer)* и *glEnd*. Константы, передаваемые команде *glBegin* в качестве параметра, определяют вид рисуемых геометрических примитивов. В таблице ниже перечислены некоторые из них.

Значение параметра mode	Описание	Изображение
GL_POINTS	Рисуются отдельные точки	
GL_LINES	Каждая пара вершин задает отрезок	
GL_TRIANGLES	Каждая тройка вершин задает отдельный треугольник	
GL_TRIANGLE_STRIP	Рисуются треугольники с общей стороной	
GL_QUADS	Каждые четыре вершины задают четырехугольник	

В примере ниже рисуются два треугольника красного и зеленого цвета.


```

glBegin(GL_TRIANGLES);

    //нарисуем красный треугольник
    glColor3f(1, 0, 0);
    glVertex2d(-0.8, -0.8);
    glVertex2d(-0.1, -0.8);
    glVertex2d(-0.8, 0.8);

    //нарисуем зеленый треугольник
    glColor3f(0, 1, 0);
    glVertex2d(0.8, 0.8);
    glVertex2d(0.1, 0.1);
    glVertex2d(0.8, 0.1);

glEnd;

```

Использование массивов вершин и нормалей

В OpenGL, начиная с версии 1.1, предусмотрена возможность передавать в графический конвейер данные массивами. Для перехода к использованию массивов вершин и нормалей нужно вызвать команду *glEnableClientState(array : integer)* по очереди с параметрами *GL_VERTEX_ARRAY* и *GL_NORMAL_ARRAY*. Для отключения этого режима используется команда *glDisableClientState(array : integer)*, которой передаются те же самые константы. Затем нужно установить указатели на массивы при помощи команд *glVertexPointer(size, type, stride : integer; ptr : pointer)* и *glNormalPointer(type, stride : integer; ptr : pointer)*. Параметр *size* определяет количество координат у вершин. У нормалей всегда три координаты. Параметр *type* задает тип массива: *GL_FLOAT*, *GL_DOUBLE* и т.п.. Параметр *stride* должен быть равен 0, если вершины расположены в памяти одна за другой. Если между вершинами находятся другие данные, то этот параметр должен быть равен смещению в байтах от начала одной вершины до начала следующей за ней. Через параметр *ptr* передается указатель на начало массива координат.

Для рисования объектов, заданных массивами, используются команды *glArrayElement*, *glDrawElements*, *glDrawArrays* и др.

Команда *glArrayElement(index : integer)* рисует одну вершину. Для каждого разрешенного командой *glEnableClientState* массива она вызывает соответствующую команду, передавая ей элемент с заданным индексом. Например, для массива нормалей это будет одна из команд *glNormal*()*, для массива вершин — *glVertex*()*. Команды *glArrayElement* указываются между

glBegin и *glEnd*. Ниже приводится пример рисования треугольника с индексами вершин и нормалей 0, 10 и 20.

```

Var
  //массив координат вершин
  V : array [0..99] of TVertex = (...);

  //массив координат нормалей
  N : array [0..99] of TVector3 = (...);

  .....

//разрешим использование массива вершин
glEnableClientState(GL_VERTEX_ARRAY);

//определим указатель на массив вершин
glVertexPointer(4, GL_DOUBLE, 0, @V);

//разрешим использование массива нормалей
glEnableClientState(GL_NORMAL_ARRAY);

//определим указатель на массив нормалей
glNormalPointer(GL_DOUBLE, 0, @N);

glBegin(GL_TRIANGLES);
  //нарисуем треугольник с вершинами и нормальями 0, 10 и 20
  glArrayElement(0);
  glArrayElement(10);
  glArrayElement(20);
glEnd;

//запретим использование массивов
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);

```

Команда *glDrawElements(mode, count, type : integer; indices : pointer)* позволяет рисовать объекты целиком. Параметр *mode* принимает значения *GL_TRIANGLES*, *GL_QUADS* и т.п. и определяет, из каких элементов состоит поверхность объекта. Параметр *count* должен быть равен количеству вершин объекта. Параметр *type* принимает значение *GL_UNSIGNED_BYTE*, *GL_UNSIGNED_SHORT* или *GL_UNSIGNED_INT* и указывает тип индексов вершин. Через параметр *indices* передается указатель на массив индексов. Вызов этой функции аналогичен вызову *glArrayElement* для каждого индекса в массиве *indices*. В приведенном ниже примере с помощью команды *glDrawElements* рисуется пирамидка, показанная на рис. 8. Предполагается,

что указатели на массивы вершин и нормалей уже установлены и их использование разрешено. Массив F содержит индексы вершин и нормалей каждой грани.

```
//индексы вершин и нормалей граней пирамидки
F : array[0..3, 0..2] of integer = (
  (0, 1, 2),
  (0, 2, 3),
  (1, 3, 2),
  (0, 3, 1));

.....

//нарисуем пирамидку
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_INT, @F);
```

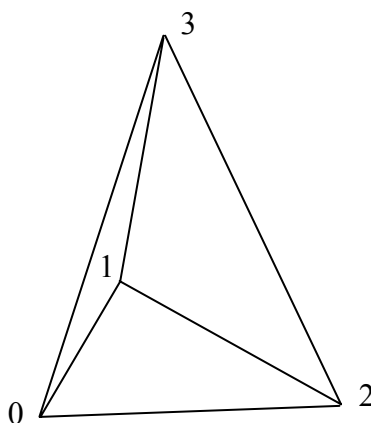


Рис. 8. Пример полигональной поверхности

Функция `glDrawArrays(mode, first, count : integer)` также, как и `glDrawElements`, позволяет нарисовать за один раз большое число геометрических примитивов. Но, в отличие от последней, она не использует дополнительного массива индексов и выбирает вершины и нормали последовательно. Например, для пирамидки с рис. 8 и массив вершин, и массив нормалей должны содержать по 12 элементов: по три на каждую грань. Параметр `mode` принимает значения `GL_TRIANGLES`, `GL_QUADS` и т.п., параметр `first` должен быть равен индексу первой вершины, с которой начнется рисование, параметр `count` — их числу.

Начиная с версии OpenGL 1.5 массивы данных можно помещать непосредственно в видеопамять, что дает выигрыш в производительности.

Рисование прозрачных объектов

Прозрачность в OpenGL моделируется с помощью смешивания цветов. Например, если цвет точки в цветовом буфере равен C_d , а цвет рисуемой поверх нее точки равен C_s , то добиться эффекта прозрачности можно, объединив эти цвета следующим образом: $C = C_s \cdot \alpha + C_d \cdot (1 - \alpha)$, где α — коэффициент непрозрачности рисуемого объекта. Он задается для материала вместе с коэффициентами диффузного отражения света как последняя четвертая компонента (подробности см. в разделе «Освещение и материалы»). Возможны и другие варианты смешивания. Параметры команды *glBlendFunc(sfactor, dfactor : integer)* определяют коэффициенты, на которые будет умножаться соответственно цвет рисуемых точек и цвет точек из буфера. Для того, чтобы получить смешивание по вышеприведенной формуле, нужно использовать вызов *glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)*. Включение режима смешивания цветов выполняется командой *glEnable(GL_BLEND)*, выключение — *glDisable(GL_BLEND)*.

Для того, чтобы прозрачные объекты выглядели правильно, их нужно выводить в порядке от самого дальнего до самого ближнего. Этого можно добиться, например, сортировкой граней по удалению от камеры. Однако, если все объекты имеют практически одинаковый цвет, то разница будет малозаметна. Единственное, что нужно здесь сделать — это отключить запись в z-буфер. В противном случае дальние объекты не будут нарисованы после ближних. Отключение записи в z-буфер выполняется командой *glDepthMask(false)*, включение — *glDepthMask(true)*.

Нужно заметить, что прозрачные объекты всегда следует рисовать в последнюю очередь. Иначе не получится эффекта прозрачности. Если данные модели организованы таким образом, что отделить прозрачные элементы от непрозрачных затруднительно, то можно воспользоваться функцией альфа-теста. Она позволяет рисовать элементы только с определенным значением коэффициента α . Сцена рисуется два раза. В первый раз нужно разрешить рисование только непрозрачных объектов (с коэффициентом $\alpha = 1$), во второй — только прозрачных ($\alpha < 1$). Альфа-тест включается командой *glEnable(GL_ALPHA_TEST)*, выключается командой *glDisable(GL_ALPHA_TEST)*. Функция сравнения задается командой *glAlphaFunc(func, ref : integer)*. Первый параметр может принимать значения *GL_EQUAL*, *GL_LESS*, *GL_GREATER* и т.п., второй задает эталонное значение, с которым будет сравниваться коэффициент непрозрачности α . Например, чтобы разрешить рисование только непрозрачных объектов,

нужно использовать команду *glAlphaFunc(GL_EQUAL, 1)*, для рисования только прозрачных — *glAlphaFunc(GL_LESS, 1)*. Следующий пример демонстрирует вышеописанный подход.

```
//включим альфа-тест
glEnable(GL_ALPHA_TEST);

//разрешим рисование только непрозрачных объектов
glAlphaFunc(GL_EQUAL, 1);

//нарисуем всю сцену
.....

//разрешим рисование только прозрачных объектов
glAlphaFunc(GL_LESS, 1);

//включим смешивание цветов
glEnable(GL_BLEND);

//определим формулу смешивания
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

//запретим запись в z-буфер
glDepthMask(false);

//нарисуем всю сцену
.....

//разрешим запись в z-буфер
glDepthMask(true);

//отключим смешивание цветов
glDisable(GL_BLEND);

//отключим альфа-тест
glDisable(GL_ALPHA_TEST);
```

7. Задание

1. Создать с помощью 3DMax модель в соответствии с выбранным вариантом задания. Модель должна включать в себя один или несколько зависимых объектов (пропеллер самолета и т.п.). Для каждого объекта должен быть задан соответствующий материал.
2. Создать описание модели в структурах данных на одном из языков программирования.
3. Создать программу, управляющую данной моделью. При перемещении и повороте независимых объектов зависимые должны перемещаться и поворачиваться вместе с ними.

Варианты моделей

1. Работающий настольный вентилятор, медленно поворачивающийся вокруг вертикальной оси.
2. Катящееся по доске туда и обратно колесо. Доска должна качаться, не давая колесу скатиться.
3. Изображение компьютера на вращающемся столе.
4. Холодильник с открывающейся и закрывающейся дверцей.
5. «Рука» робота с двумя суставами.
6. Стол с выдвижающимися ящиками.
7. Избушка с трубой и открывающейся дверью и окошками.
8. Открывающийся и закрывающийся сундук.
9. Движущийся по плоскости трактор.
10. Молоток, бьющий по гвоздю.
11. Башня замка с подъемным мостом надо рвом.
12. Работающая водяная мельница.
13. Работающий разводной мост над каналом.
14. Гардероб с открывающимися дверцами и выдвижающимися ящиками.
15. Движущийся БТР с вращающейся башней.
16. Газовая плита с открывающейся дверцей.
17. Управляемый колесный погрузчик.
18. Движущийся самосвал с поднимающимся и опускающимся кузовом.
19. Работающий экскаватор.
20. Поднимающийся и опускающийся лифт с открывающейся дверью.
21. Работающий колодец с ведром.
22. Летящий пароплан с пропеллером.
23. Летящий дирижабль с пропеллером.
24. Движущаяся по круглой трассе машина.
25. Работающая радарная установка.