**Основы программирования OpenGL в Borland С++Builder и Delphi. Простейшие объекты**

Луковкин Сергей

Рассматривая какой-либо трёхмерный объект, мы всегда определяем его положение и размеры относительно некоторой привычной, и удобной в настоящий момент системы координат, связанной с реальным миром. Такая исходная система координат в компьютерной графике является правосторонней и называется мировой системой координат....

Для того, чтобы можно было изобразить объект на экране, его необходимо предварительно перевести (или преобразовать) в другую систему координат, которая связана с точкой наблюдения и носит название видовой системы координат. Эта система координат является левосторонней. И, наконец, любое трёхмерное изображение мы всегда рисуем на двумерном экране, который имеет свою экранную систему координат. (Этот абзац я списал у Ю.Тихомирова).

|  |  |
| --- | --- |
| Правосторонняя система координат (мировая) | Левосторонняя система координат (видовая) |
|  |  |

По умолчанию, плоскость xOy параллельна экрану, а ось Z направлена в мировых координатах к нам, в видовых - от нас.

**Переход к новым координатам**

В OpenGL все объекты рисуются в начале координат, т.е. в точке (0,0,0). Для того, чтобы изобразить объект в точке (x1,y1,z1), надо переместить начало координат в эту точку, т.е. перейти к новым координатам. Для этого в OpenGL определены две процедуры:

glTranslate[f d](Dx, Dy, Dz) – сдвигает начало координат на (Dx, Dy, Dz)

glRotate[f d](j, x,y,z) – поворачивает систему координат на угол j (в градусах) против часовой стрелки вокруг вектора (x,y,z)

ПРИМЕЧАНИЕ: [f d] – означает, что в конце может быть либо буква “f”, либо “d”.

Теперь стоит сказать ещё о двух процедурах:

glPushMatrix

glPopMatrix

Первая предназначена, для сохранения, а вторая – для восстановления текущих координат. Очень удобно с помощью glPushMatrix сохранить текущие координаты, потом сдвигаться и вертеться как угодно, а после, вызовом glPopMatrix, вернуться к исходным координатам. Параметров у этих процедур нет.

**Часть 2. Простейшие фигуры**

**Простейшие объёмные фигуры**

В примере из прошлой статьи мы создали сферу. Для этого мы использовали механизм из glu32.dll. Алгоритм был такой:

1. Создаём объект типа GLUquadricObj

2. Инициализируем его функцией gluNewQuadric

3. Устанавливаем стиль фигуры функцией gluQuadricDrawStyle(quadObj, GLU\_FILL). Стиль может быть GLU\_FILL, GLU\_LINE, GLU\_SILHOUETTE или GLU\_POINT. Что каждый из них значит, проверьте сами.

4. Делаем из quadObj (объекта типа GLUquadricObj) сферу, цилиндр, конус, диск или часть диска. Для этого определены следующие функции:

· gluSphere (quadObj, radius, slices, loops). Три последних параметра – это радиус и количество разбиений поперёк и вдоль оси Z соответственно.

· gluCylinder (quadObj, baseRadius, topRadius, height, slices, loops). После quadObj идут следующие параметры: радиус нижнего основания, радиус верхнего основания, высота и количество разбиений поперёк и вдоль оси Z соответственно. Очевидно, что эта функция задаёт как цилиндр, так и конус.

· gluDisk (quadObj, innerRadius, outerRadius, slices, loops). Здесь после quadObj указываются внутренний и внешний радиусы диска.

· gluPartialDisk (quadObj, innerRadius, outerRadius, slices, loops, startAngle, sweepAngle). Здесь добавляются два параметра: угол (в градусах), с которого начнётся рисование диска, и угол, которым рисование закончится.

5. Освобождаем память, занимаемую под quadObj функцией gluDeleteQuadric(quadObj).

Теперь вы можете рисовать простые трёхмерные фигуры!

**Примитивы**

Любую трёхмерную фигуру, какая бы сложная она не была, можно разбить на двухмерные (плоские) составляющие. Эти составляющие я и буду называть примитивами, хотя некоторые авторы считают, что примитивами следует обозвать вышеперечисленные трёхмерные фигуры.

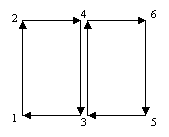
Примитивы определяются одной или несколькими точками, которые в OpenGL задаются внутри командных скобок glBegin/glEnd:

|  |
| --- |
| С++ |
| void glBegin(mode);  void glEnd(); |
| Delphi |
| procedure glBegin(mode);  procedure glEnd; |

Параметр mode показывает, какие примитивы будут рисоваться. Доступны следующие значения:

|  |  |
| --- | --- |
| GL\_POINTS | Каждая вершина – отдельная точка |
| GL\_LINES | Каждая пара вершин – отдельная линия. Если число вершин нечётно, то последняя игнорируется |
| GL\_LINE\_STRIP | Последовательность связанных отрезков. Первые две вершины – первый отрезок. Третья вершина определяет второй отрезок с началом в конце первого и концом в этой вершине и т.д |
| GL\_LINE\_LOOP | Аналогичен GL\_LINE\_STRIP, только последняя вершина соединяется отрезком с первой. |
| GL\_TRIANGLES | Каждая тройка вершин – отдельный треугольник |
| GL\_TRIANGLE\_STRIP | Группа связанных треугольников. Первые три вершины – первый треугольник. Вторая, третья и четвёртая вершины - второй треугольник и т.д. |
| GL\_TRIANGLE\_FAN | Также группа связанных треугольников. Первые три вершины – первый треугольник. Первая, вторая и четвёртая вершины - второй треугольник и т.д. |
| GL\_QUADS | Каждые четыре вершины – отдельный четырёхугольник. |
| GL\_QUAD\_STRIP | Группа связанных четырёхугольников. Первые четыре вершины – первый четырёхугольник. Третья, четвёртая, пятая и шестая вершины – второй четырёхугольник и т.д. |
| GL\_POLYGON | Рисует отдельный выпуклый многоугольник (один). |

Особое внимание нужно уделить GL\_QUAD\_STRIP. Здесь не совсем понятный, но очень удобный порядок указания вершин:



У каждого примитива есть минимальное число вершин. Если указанное число вершин меньше минимального для данного примитива, то примитив не рисуется.

Осталось только сказать, как задать вершину. Для этого определена следующая процедура:

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

Вершина определяется четырьмя параметрами: координаты x, y, z и параметр w – коэффициент, на который делится каждая из координат, т.е. w определяет масштаб. По умолчанию z=0, w=1, т.е когда вы вызываете, например, glVertex2f(1,1) на самом деле вызывается glVertex4f(1,1,0,1).

С каждой вершиной связаны некоторые данные:

· Текущий цвет – цвет вершины (окончательный цвет высчитывается с учётом света). Цвет задаётся процедурой glColor\*

· Текущие координаты текстуры – координаты текстуры, соответствующие этой вершине. Задаются процедурой glTexCoord\*

· Текущая нормаль – вектор нормали, соответствующий данной вершине. Задаётся процедурой glNormal\*

· Текущая позиция растра – используется для определения положения растра при работе с пикселями и битовыми массивами. задаётся процедурой glRasterPos\*

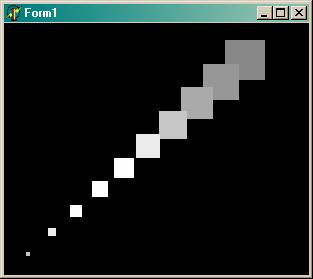
ПРИМЕЧАНИЕ: вместо звёздочки ‘\*’ ставятся соответствующие суффиксы; такое сокращение принято во многих документациях по OpenGL.

**Точки**

Нарисовать точку очень просто. Следующий код изображает 10 точек разного размера.

|  |
| --- |
| С++ |
| void TForm1::Draw()  {  glClear(GL\_DEPTH\_BUFFER\_BIT or GL\_COLOR\_BUFFER\_BIT);  glColor3f(1,1,1);  Byte i;  for(i=0;i<10;i++)  {  glPointSize((i+1)\*4);  glBegin(GL\_POINTS);  glVertex2f(i,i);  glEnd();  }  SwapBuffers(ghDC);  } |
| Delphi |
| procedure TForm1.Draw;  var  i:byte;  begin  glClear(GL\_DEPTH\_BUFFER\_BIT or GL\_COLOR\_BUFFER\_BIT);  glColor3f(1,1,1);  for i:=0 to 9 do  begin  glPointSize((i+1)\*4);  glBegin(GL\_POINTS);  glVertex2f(i,i);  glEnd;  end;  SwapBuffers(ghDC);  end; |

ПРИМЕЧАНИЕ: в FormResize я вызвал glOrtho следующим образом - glOrtho(-1,12, -1,12, 2,12). Это – для того, чтобы все точки поместились в окне.

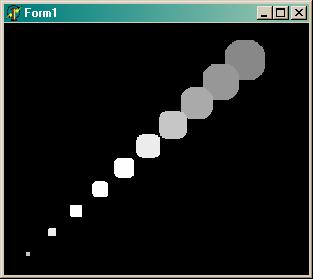


Для изменения размера точки используется процедура glPointSize(size). Параметр size задаёт диаметр точки.

В этом примере все точки квадратные. В OpenGL разрешено сглаживание (smoothening) как точек, так и более сложных объектов. Как и всё в OpenGL, этот режим включается и выключается процедурами glEnable/glDisable. Для точек это делается так:

glEnable(GL\_POINT\_SMOOTH);

Вставив эту строчку где-нибудь перед рисованием точек, получим:



Откровенно говоря, у меня OpenGL делает это довольно плохоL, возможно ваша реализация справляется с этим лучше.

**Линии**

С линиями – не на много сложнее. Вместо размера у линии указывается ширина:

glLineWidth(width)

сглаживание разрешается следующим образом:

glEnable(GL\_LINE\_SMOOTH)

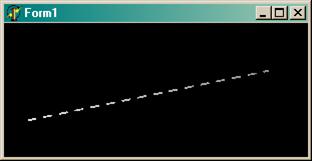
Но на этом возможности линий не заканчиваются. Я уже рассказал, как можно нарисовать две или даже три линии, указав всего три вершины (вызываем glBegin с параметром GL\_LINE\_STRIP или GL\_LINE\_LOOP), но и это ещё не всё! В OpenGL можно указать штриховку линии! Делается это процедурой glLineStipple(factor, pattern). Здесь pattern – 16-разрядная битовая маска. Например, чтобы нарисовать пунктирную линию, маску надо задать равной 255, что в шестнадцатеричной системе счисления соответствует 00FF, а в двоичной – 0000000011111111. А целое factor показывает, сколько раз будет повторяться каждый бит маски.

Осталось только разрешить штриховать линии: glEnable(GL\_LINE\_STIPPLE).

Пример.

|  |
| --- |
| С++, Delphi |
| glEnable(GL\_LINE\_SMOOTH);  glLineStipple(1,255);  glEnable(GL\_LINE\_STIPPLE);  glBegin(GL\_LINES);  glVertex2f(0,2);  glVertex2f(10,6);  glEnd; |

Вот, что получится:



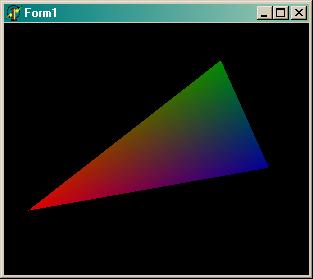
**Полигоны**

Теперь перейдём к плоским фигурам: треугольникам, четырёхугольникам и произвольным выпуклым многоугольникам. С ними можно делать всё то же, что и с линиями (только сглаживание включается и выключается процедурами glEnable/glDisable с параметром GL\_POLYGON\_SMOOTH), плюс ещё одна процедура: glPolygonMode(face, mode). Второй параметр – mode – указывает, как будет рисоваться полингон (по русски - многоугольник). Он может принимать значения GL\_POINT(рисуются только точки), GL\_LINE(только линии) или GL\_FILL(заполненный полигон). А первый параметр – face – показывает, какой стороне полигона применяется режим mode: GL\_FRONT(к лицевой), GL\_BACK(к тыльной) или GL\_FRONT\_AND\_BACK(к обеим).

Давайте нарисуем треугольник. Вот как будет выглядеть функция Draw:

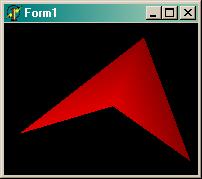
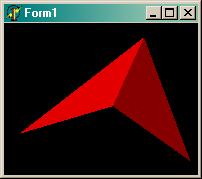
|  |
| --- |
| С++ |
| void TForm1::Draw()  {  glClear(GL\_DEPTH\_BUFFER\_BIT or GL\_COLOR\_BUFFER\_BIT);  glPolygonMode(GL\_FRONT\_AND\_BACK,GL\_FILL);  glBegin(GL\_TRIANGLES);  glColor3f(1,0,0);  glVertex2f(0,2);  glColor3f(0,1,0);  glVertex2f(8,9);  glColor3f(0,0,1);  glVertex2f(10,4);  glEnd();  SwapBuffers(ghDC);  } |
| Delphi |
| procedure TForm1.Draw;  begin  glClear(GL\_DEPTH\_BUFFER\_BIT or GL\_COLOR\_BUFFER\_BIT);  glPolygonMode(GL\_FRONT\_AND\_BACK,GL\_FILL);  glBegin(GL\_TRIANGLES);  glColor3f(1,0,0);  glVertex2f(0,2);  glColor3f(0,1,0);  glVertex2f(8,9);  glColor3f(0,0,1);  glVertex2f(10,4);  glEnd;  SwapBuffers(ghDC);  end; |

Я уже говорил, что каждая вершина может иметь свой цвет, этим я здесь и воспользовался. И вот что получилось:



Каждой вершине указывать цвет совсем не обязательно. Если вы хотите нарисовать треугольник одного цвета, то этот цвет указывается один раз – перед рисованием самого примитива.

Забегая вперёд, скажу, что плавного перетекания цветов как на рисунке может и не быть, если перед рисованием вызвать процедуру glShadeModel(GL\_FLAT), по умолчанию её параметр – GL\_SMOOTH. Эта процедура указывает, сглаживать или нет углы между смежными полигонами. Вот картинки для иллюстрации её действия:

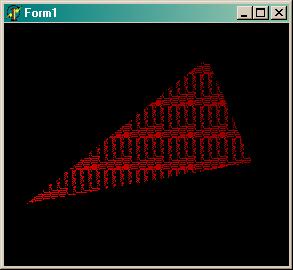


Раз уж я сказал о штриховке линий, то нужно сказать и о трафарете – штриховке для полигонов. Он включается командой glEnable(GL\_POLYGON\_STIPPLE). Также как и с линиями, трафарет задаётся массивом, который определяет битовую маску. Размер трафарета - 32x32 бита, т.е. размер массива будет 128 байт.

Мне было лень прописывать каждый из 128 байт маски по отдельности, и я сформировал её в цикле, и вот результат:

|  |
| --- |
| С++ |
| void TForm1::Draw()  {  glClear(GL\_DEPTH\_BUFFER\_BIT or GL\_COLOR\_BUFFER\_BIT);  // формируем маску  for(int k=0;k<16;k++)  for(int i=0;i<8;i++)  stip[k][i]:=k-i;  glEnable(GL\_POLYGON\_STIPPLE);  glPolygonStipple(@stip);  glPolygonMode(GL\_FRONT\_AND\_BACK,GL\_FILL);  glColor3f(1,0,0);  glBegin(GL\_TRIANGLES);  glVertex2f(0,2);  glVertex2f(8,9);  glVertex2f(10,4);  glEnd();  SwapBuffers(ghDC);  } |
| Delphi |
| procedure TForm1.Draw;  var  stip:array [1..16,1..8] of GLubyte;  i,k:byte;  begin  glClear(GL\_DEPTH\_BUFFER\_BIT or GL\_COLOR\_BUFFER\_BIT);  // формируем маску  for k:=1 to 16 do  for i:=1 to 8 do  stip[k][i]:=k-i;  glEnable(GL\_POLYGON\_STIPPLE);  glPolygonStipple(@stip);  glPolygonMode(GL\_FRONT\_AND\_BACK,GL\_FILL);  glColor3f(1,0,0);  glBegin(GL\_TRIANGLES);  glVertex2f(0,2);  glVertex2f(8,9);  glVertex2f(10,4);  glEnd;  SwapBuffers(ghDC);  end; |

Вот результат:



Вообще маска формируется один раз, поэтому, если вы перенесёте код её формирования в другое место (например в FormCreate), то программа будет работать быстрее.

Ещё хотелось бы сказать о массивах OpenGL. Этот метод позволяет хранить все вершины объекта в массиве, причем в этом массиве можно хранить не только координаты вершин, но и их атрибуты (иногда это бывает полезно). Но товарищи из Borland’а решили, что нам это не нужно и не объявили соответствующие процедуры и константы. Без этого легко можно обойтись, но всё-таки обидноL.

**Параллелепипед**

Не знаю, заметили вы или нет, но GLU не позволяет создавать параллелепипеды. Давайте это исправим: напишем процедуру, рисующую параллелепипед.

|  |
| --- |
| С++ |
| void piped(GLfloat a, GLfloat b, GLfloat c)  {  glShadeModel(GL\_FLAT);  glPolygonMode(GL\_FRONT\_AND\_BACK,GL\_FILL);  glBegin(GL\_QUAD\_STRIP);  glVertex3f(-a/2,-b/2,-c/2);  glVertex3f(-a/2,-b/2, c/2);  glVertex3f(-a/2, b/2,-c/2);  glVertex3f(-a/2, b/2, c/2);  glVertex3f( a/2, b/2,-c/2);  glVertex3f( a/2, b/2, c/2);  glVertex3f( a/2,-b/2,-c/2);  glVertex3f( a/2,-b/2, c/2);  glVertex3f(-a/2,-b/2,-c/2);  glVertex3f(-a/2,-b/2, c/2);  glEnd();  glBegin(GL\_QUADS);  glVertex3f(-a/2,-b/2, c/2);  glVertex3f(-a/2, b/2, c/2);  glVertex3f( a/2, b/2, c/2);  glVertex3f( a/2,-b/2, c/2);  glVertex3f(-a/2,-b/2,-c/2);  glVertex3f(-a/2, b/2,-c/2);  glVertex3f( a/2, b/2,-c/2);  glVertex3f( a/2,-b/2,-c/2);  glEnd();  } |
| Delphi |
| procedure piped(a,b,c:GLfloat);  begin  glShadeModel(GL\_FLAT);  glPolygonMode(GL\_FRONT\_AND\_BACK,GL\_FILL);  glBegin(GL\_QUAD\_STRIP);  glVertex3f(-a/2,-b/2,-c/2);  glVertex3f(-a/2,-b/2, c/2);  glVertex3f(-a/2, b/2,-c/2);  glVertex3f(-a/2, b/2, c/2);  glVertex3f( a/2, b/2,-c/2);  glVertex3f( a/2, b/2, c/2);  glVertex3f( a/2,-b/2,-c/2);  glVertex3f( a/2,-b/2, c/2);  glVertex3f(-a/2,-b/2,-c/2);  glVertex3f(-a/2,-b/2, c/2);  glEnd;  glBegin(GL\_QUADS);  glVertex3f(-a/2,-b/2, c/2);  glVertex3f(-a/2, b/2, c/2);  glVertex3f( a/2, b/2, c/2);  glVertex3f( a/2,-b/2, c/2);  glVertex3f(-a/2,-b/2,-c/2);  glVertex3f(-a/2, b/2,-c/2);  glVertex3f( a/2, b/2,-c/2);  glVertex3f( a/2,-b/2,-c/2);  glEnd;  end; |

Можно проверять!

glOrtho вызовем также, как и в примере со сферой: glOrtho(-5,5, -5,5, 1,12).

А в Draw напишем следующее:

|  |
| --- |
| С++, Delphi |
| glColor3f(0.6,0.7,0.9);  glPushMatrix;  glRotatef(10, 0,0,1);  glRotatef(25, 0,1,0);  glRotatef(20, 1,0,0);  piped(5,1.2,3.5);  glPopMatrix; |

И получим картинку:

