OpenGL Course: Part 2
POINTS, LINES AND POLYGONS

THOMAS G. E. RUGE, PABLO GUSSMANN

This part of the OpenGL course firstly concerns the basic graphical elements from which 3D objects are constructed. It will also explain how the objects created can be correctly lit.

Before we come to the basic elements, the co-ordinate system used by OpenGL must first be explained in more detail. This is a Cartesian co-ordinate system.

The x and y-axes form a plane, something like the visible surface of a monitor. The z-axis adds the third dimension - spatial depth.

In the case of our monitor this would now be the depth of the picture tube. A point P thus needs three values (x,y,z) in order to have a fixed position in our co-ordinate system.

Basic 3D elements

As demonstrated in the first part, the general command structure of OpenGL in order to draw something looks something like this:

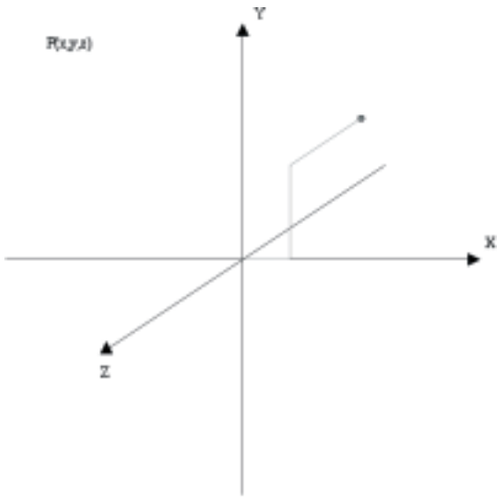
```
glBegin(...);
glColor3f(..);
glVertex3f(...);
glColor3f(..);
glVertex3f(...);
...
glEnd();
```

glBegin(TYPE) tells the machine which basic element (also referred to as a primitive) it should draw from now on. A complete illustration of all OpenGL primitives can be seen in Figure 2.

In OpenGL there are five different basic elements, from which all objects must be composed. In detail, these are points, lines, triangles, quadrangles and polygons. Variations can be formed out of all these elements (apart from points), mostly simple continuations as the result of defining additional vertices. So a simple triangle can turn into a so-called TRIANGLE_STRIP or a quadrangle (GL_QUAD) can become a QUAD_STRIP (see Figure 2).

This has the additional advantage that the overlapping vertices in the composed element do not have to be loaded into memory and calculated twice. So if we want to draw four coherent triangles, it is sufficient to specify six points - saving three sides and six points.

Figure 1: The Cartesian co-ordinate system



Colours

A colour in OpenGL is normally based on the RGB principle, thus it consists of the components Red, Green and Blue. So all visible colours from Black to White can be mixed.

Examples of OpenGL colours	
Green:	*glColor3f(0.0f, 1.0f, 0.0f);*
Violet:	*glColor3f(0.6f, 0.0f, 0.4f);*
Black:	*glColor3f(0.0f, 0.0f, 0.0f);*
Grey:	*glColor3f(0.4f, 0.4f, 0.4f);*
White:	*glColor3f(1.0f, 1.0f, 1.0f);*

The first example shows all 10 of the primitive types mentioned above.

The second example from the last part of the course serves as the basis for this. So it is again based on GLUT, the OpenGL Utility Toolkit.

Let there be light

Objects such as the teapot from the first part consist of triangles

or polygons. To make them look more realistic, these must be lit and thus appear brighter or darker, depending on the angle formed between them and the source of light. OpenGL fortunately takes over this part of the maths for us, but it still requires additional information. And this is in the form of normal vectors, thus a vector that stands vertically to a surface.

Figure 3 shows an image of a normal vector on an area. An area consists of at least three points. The vectors u and v are the vectors from P1 to P2, and P1 to P4 respectively. But it doesn't matter which point is used to form u and v, since if the points are correctly oriented with respect to the area, the normal vector is always the same.

The cross product

The normal vectors of an area can be calculated using the cross product.

```
vNorm.x = u.y * v.z - u.z * v.y
vNorm.y = u.z * v.x - u.x * v.z
vNorm.z = u.x * v.y - u.y * v.x
```

So that the normal vector is also pointing in the right direction, it is again necessary for the sequence of points (P1, P2, P3...) which define the area to be consistent.

But this leaves the normal vector still only half-finished, because it still has to be standardised. This is necessary to ensure that all normal vectors have the same length. This then looks something like this:

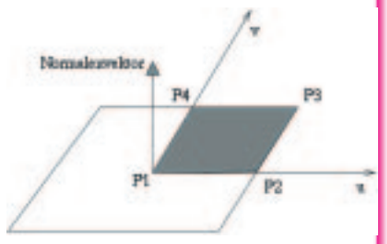
```
length = sqrt( vNorm.x * vNorm.x + vNorm.y * ➥
vNorm.y + vNorm.z vNorm.z)
vNorm.x /= length
vNorm.y /= length
vNorm.z /= length
```

This should, of course, not be calculated anew for each frame because the computing time taken would be enormous. These normal vectors should be calculated just once at the start of the program, because they do not change (in most cases).

Using *glNormal3f(vNorm.x, vNorm.y, vNorm.z);* these values are transferred to OpenGL, so this is just the same as with colour values.

Light in OpenGL

Of course, in order to show illumination with normal vectors, you also need a source of light. For this we need a few details about its position and



[left]
Figure 2: OpenGL primitives

[right]
Figure 3: Normal vectors

colour values (the light source need not, of course, only give out white light). The following variables contain the necessary values for the position of the light source:

```
GLfloat LightPosition[] = \
{0.0, 0.0, -1.0, 1.0f };
```

The first three values specify the position and the fourth is a sort of switch, which should stay at 1.0.

The next two variables contain the values for the ambient and the diffuse components of the light:

```
GLfloat LightAmbient[] = \
{0.2, 0.2f, 0.2f, 1.0f };
GLfloat LightDiffuse[] = \
{0.3f, 0.3f, 0.3f, 1.0f };
GLfloat LightSpecular[] = \
{.9f, .9f, .9f, 1.0f };
```

Listing 1, Primitives.c

The program is compiled with:

```
gcc -I . -c Primitives.c
gcc -o Primitives Primitives.o \
-L /usr/X11R6/lib/ -lGL -lglut -lGLU
```

The program is really very simple to explain: You can select the type of primitive using keys 1..9. This is done in the callback function (see Part 1) for drawing. Primitives are always drawn with different colours. The case query in the keyboard callback sets the value for the primitives in the variable *draw_type*, which is then queried in turn in callback for the drawing. The following commands from the program draw a red triangle.

```
glBegin(GL_TRIANGLES);
glColor3f(1.0f, 0.0f, 0.0f);
glVertex3f(-100.0f, 0.0f, -100.0f);
glVertex3f(-100.0f, 100.0f, -100.0f);
glVertex3f(0.0f,100.0f, -100.0f);
glEnd();
```

Most routines have been taken over entirely or expanded from the last part of the course. The program run has remained the same. During navigation it sometimes happens that when surfaces are drawn they are not always visible. This happens if the surface is turned away from the onlooker. Normally the sequence of vertices of polygons is defined uniformly, clockwise or anticlockwise.

This is prevented by the command *glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)*. So both sides of the polygons are declared as visible.

Vector	
Forwards:	*glNormal3f(0.0f, 0.0f, 1.0f);*
Backwards:	*glNormal3f(0.0f, 0.0f, -1.0f);*
Right:	*glNormal3f(1.0f, 0.0f, 0.0f);*
Left:	*glNormal3f(-1.0f, 0.0f, 0.0f);*
Up:	*glNormal3f(0.0f, 1.0f, 0.0f);*
Down:	*glNormal3f(0.0f, -1.0f, 0.0f);*

Here, the first three values specify the Red Green Blue (RGB) values at which the light should shine.

The **ambient** component of the light is the part of the light that comes from no particular direction. It arises, for example, when light falls into a space and the rays strike everywhere and are reflected until they are no longer coming from any definable direction and are only present in the form of background light.

The **diffuse** portion of the light comes from a specific direction and is reflected evenly over an area. Areas which are tilted towards the light source appear brighter than those turned away from the light.

The **specular** part of the light also comes, like the diffuse part of the light, from one direction but is reflected unevenly over an area. As the result of this, bright spots of light are created on surfaces.

These values are now allocated as follows to the light source GL_LIGHT0:

```
glLightfv(GL_LIGHT0, \
GL_AMBIENT, LightAmbient);
glLightfv(GL_LIGHT0, \
GL_DIFFUSE, LightDiffuse);
glLightfv(GL_LIGHT0, \
GL_SPECULAR, LightSpecular);
glLightfv( GL_LIGHT0, \
GL_POSITION, LightPosition );
```

Using

```
glEnable(GL_LIGHT0);
```

the light source and with

```
glEnable(GL_LIGHTING);
```

the lighting calculation are started by OpenGL. Now the area no longer appears just in the full colour, but brighter or darker, depending on how they stand with respect to the light. OpenGL provides a maximum of eight light sources. These can have different colours and positions and be switched on or off.

So that the light source also works on coloured areas (only very few are white), OpenGL still has to be instructed on how to apply the light calculation to the colour values of areas.

```
glEnable ( GL_COLOR_MATERIAL );
glColorMaterial ( GL_FRONT_AND_BACK, GL_AMBI➥
ENT_AND_DIFFUSE );
```

Below is a short sample program, which draws, lights and rotates a dice made of GL_QUADS. Using 'l' and 'o' the light source can be turned on and off. Rotation can also be modified: 's' for stop and 'g' to continue rotating. ■

Info

OpenGL-Homepage: http://www.opengl.org
OpenGL and GLUT information: http://www.xmission.com/~nate/opengl.html

■

Listing 2, Light.c

The program is compiled with:

```
gcc -I . -c Light.c
gcc -o Light Light.o \
-lGL -lglut -lGLU
```

The sample program draws an illuminated dice and rotates it. The light source is situated behind the (transparent) onlooker and lights the dice from the front. It is easy to see how the areas become bright and dark. The code for the set up of the light sources and of the lighting, as described above, is in myInit(). Firstly, values are defined for the position and the properties of the light source and then they are assigned to the light source. The dice is now drawn in DrawScene(). But first the representational matrix is created and translated backwards using:

```
glTranslatef(0.0f, 0.0f, -5.0f);
```

and then rotated

```
glRotatef(rtri,.0f,1.0f,0.0f);
     rtri+= .1f;
```

The dice consists of 6 GL_QUADS and is so simple that normal vectors do not have to be calculated on a large scale. These are simple vectors, which point forwards, backwards, right, left, up and down:

Each GL_QUAD is assigned a different colour, so that the sides are easy to distinguish. Obviously, a dice is not exactly a complex object, but one with 20,000 polygons would be beyond the scope of any printer. The dice serves as the basis and can be expanded with a bit of effort. But the calculation of the normal vectors should not necessarily be undertaken manually, but automated. That's something we will come to in a later installment of this course.

The plan for the next part is an explanation of the world of matrices in more detail. Then it will be possible to program more complex procedures than a mere dice rotating about its own axis.