Computer Graphics and Multimedia Systems

OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

Since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), the OpenGL Utility Toolkit (GLUT) has been created to aid in the development of more complicated three-dimensional objects such as a sphere, a torus, and even a teapot. GLUT may not be satisfactory for full-featured OpenGL applications, but it is a useful starting point for learning OpenGL.

GLUT is designed to fill the need for a window system independent programming interface for OpenGL programs. The interface is designed to be simple yet still meet the needs of useful OpenGL programs. Removing window system operations from OpenGL is a sound decision because it allows the OpenGL graphics system to be retargeted to various systems including powerful but expensive graphics workstations as well as mass-production graphics systems like video games, set-top boxes for interactive television, and PCs.

GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT routines also take relatively few parameters.

Libraries

OpenGL provides a powerful but primitive set of rendering command, and all higher-level drawing must be done in terms of these commands. There are several libraries that allow you to simplify your programming tasks, including the following:

- OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

```
#include <GL/glut.h>
```

Initialization:

glutInit(&argc, argv)

The first thing we need to do is call the glutInit() procedure. It should be called before any other GLUT routine because it initializes the GLUT library. The parameters to glutInit() should be the same as those to main(), specifically main(int argc, char** argv) and glutInit(&argc, argv), where argcp is a pointer to the program's unmodified argc variable from main. Upon return, the value pointed to by argcp will be updated, and argv is the program's unmodified argv variable from main. Like argcp, the data for argv will be updated.

glutInitDisplayMode()

The next thing we need to do is call the glutInitDisplayMode() procedure to specify the display mode for a window. You must first decide whether you want to use an RGBA (GLUT_RGBA) or color-index (GLUT_INDEX) color model. The RGBA mode stores its color buffers as red, green, blue, and alpha color components. The forth color component, alpha, corresponds to the notion of opacity. An alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 complete transparancy. Color-index mode, in contrast, stores color buffers in indicies. Your decision on color mode should be based on hardware availability and what you application requires. More colors can usually be simultaneously represented with RGBA mode than with color-index mode. And for special effects, such as shading, lighting, and fog, RGBA mode provides more flexibility. In general, use RGBA mode whenever possible. RGBA mode is the default.

Another decision you need to make when setting up the display mode is whether you want to use single buffering (GLUT_SINGLE) or double buffering (GLUT_DOUBLE). Applications that use both front and back color buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn't displayed), then causing the front and back buffers to be swapped. If you aren't using annimation, stick with single buffering, which is the default.

Finally, you must decide if you want to use a depth buffer (GLUT_DEPTH), a stencil buffer (GLUT_STENCIL) and/or an accumulation buffer (GLUT_ACCUM). The depth buffer stores a depth value for each pixel. By using a "depth test", the depth buffer can be used to display objects with a smaller depth value in front of objects with a larger depth value. The second buffer, the stencil buffer is used to restrict drawing to certain portions of the screen, just as a cardboard stencil can be used with a can of spray paint to make a printed image. Finally, the accumulation buffer is used

for accumulating a series of images into a final composed image. None of these are default buffers.

glutInitWindowSize()

We need to create the characteristics of our window. A call to glutInitWindowSize() will be used to specify the size, in pixels, of your inital window. The arguments indicate the height and width (in pixels) of the requested window. Similarly, glutInitWindowPosition() is used to specify the screen location for the upper-left corner of your initial window. The arguments, x and y, indicate the location of the window relative to the entire display.

Creating a Window

glutCreateWindow()

To actually create a window, the with the previously set characteristics (display mode, size, location, etc), the programmer uses the glutCreateWindow() command. The command takes a string as a parameter which may appear in the title bar if the window system you are using supports it. The window is not actually displayed until the glutMainLoop() is entered.

Display Function

The glutDisplayFunc() procedure is the first and most important event callback function you will see. A callback function is one where a programmer-specified routine can be registered to be called in response to a specific type of event. For example, the argument ofglutDisplayFunc() is the function that is called whenever GLUT determines that the contents of the window needs to be redisplayed. Therefore, you should put all the routines that you need to draw a scene in this display callback function.

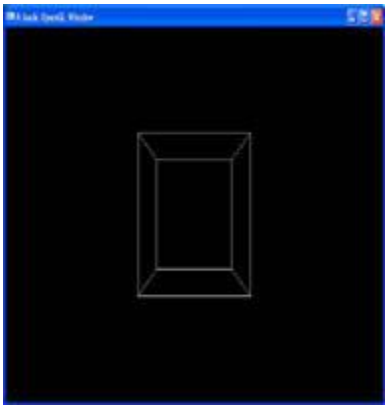
Reshape Function

The glutReshapeFunc() is a callback function that specifies the function that is called whenever the window is resized or moved. Typically, the function that is called when needed by the reshape function displays the window to the new size and redefines the viewing characteristics as desired. If glutReshapeFunc() is not called, a default reshape function is called which sets the view to minimize distortion and sets the display to the new height and width.

Main Loop

The very last thing you must do is call glutMainLoop(). All windows that have been created can now be shown, and rendering those windows is now effective. The program will now be able to handle events as they occur (mouse clicks, window resizing, etc). In addition, the registered display callback (from our glutDisplayFunc()) is triggered. Once this loop is entered, it is never exited!

How to reshape the OpenGL view correctly inside of a GLUT window?



One problem you will notice later on as you get into OpenGL, is that when your window is resized, your geometry will not transform to match the new window size. This is because when we initially setup our window, everything is setup by GLUT. But this needs to be updated every frame, or just when the window is resized.

To fix this, we create a reshape function, that will setup our OpenGL perspective matrix
according to our window size.

```
void reshape (int w, int h) {
```

To start off, we need to set the size of our viewport. Our viewport is the section of the window that is going to be rendered to. So we will set this to start at 0,0 (the top left) and it will be of the dimensions w,h (width, height). These dimensions are supplied by GLUT so we don't need to know them. Although you may find later on, that is nice to

track these variables when implementing a camera, or for using GLSL shaders.

```
glViewport (0, 0, (GLsizei)w, (GLsizei)h);
```

Once we know have set up the size of our viewport to draw to. We need to switch which mode we are in. OpenGL has many matrix modes, such as GL_PROJECTION for setting up our 'camera', GL_MODELVIEW for drawing and GL_TEXTURE for adjusting how textures are drawn onto shapes. Rish now we need to select the GL_PROJECTION mode as we are setting up our 'camera'.

```
glMatrixMode (GL_PROJECTION); //set it so we can play with the 'camera'
```

Now that we are in the mode we need, we are going to reset the matrix back to the identity matrix so that we know what we are dealing with. If we don't reset it, we can have issues occuring as we will be apply one perspective over top of a previous.

```
glLoadIdentity (); //replace the current matrix with the Identity Matrix
```

This next line, is the core of this method, it is used to setup the parameters for our camera. The first variable which I have set to 60, is the horizontal field of view (FOV). This means that we can see 30 degrees to the right, and 30 degrees to the left.

The next value is required and consists of the width divided by the height of the window. The last two values setup the near and far clipping planes. A clipping plane removes anything that is outside of it. In this case, if something is closer to the user than a unit of 1.0 on the Z axis, then it is not drawn, and if it is further than 100.0 units on the Z axis, it is also not drawn.

```
gluPerspective (60, (GLfloat)w / (GLfloat)h, 1.0, 100.0);//set the angle of view, the ratio of sight, the near and far factors
```

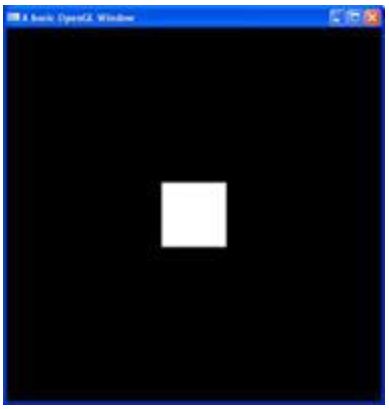
Finally, we want to go back to our GL_MODELVIEW matrix so that we can go on with any drawing that we have to do.

```
glMatrixMode (GL_MODELVIEW); //switch back the the model editing mode.
```

The last thing we need to do here, is just add one extra line of code to our main method to give our reshape method to GLUT.

```
glutReshapeFunc (reshape);
```

Creating a Square in OpenGL



OpenGL allows us to use a variety of geometrical shapes within it.

These are;
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_POINTS
GL_POLYGON
GL_QUADS
GL_QUAD_STRIP
GL_TRIANGLES
GL_TRIANGLE_FAN
GL_TRIANGLE_STRIP

In this tutorial I am only going to work with Quads to create a simple square.

As you can tell from the name Quads, we are creating a 4 sided shape.

To start of any shape, you need to set the start of it.
This is achieved with:
glBegin(shape);

So because I am using a quad, I call:
glBegin(GL_QUADS);

Next comes the code to create the vertices of the shape.
A vertex is a point in 3d space.
OpenGL will automatically link together our vertices to create our shape.

Because this is a 4 sided shape, it has 4 corners (vertices) so it creates a new shape every 5th vertex.

To set up a vertex, you can either use, 2d or 3d space.
To set up a 2d vertex, call:
glVertex2d
This takes the parameters of:
x
y
In that order.

But we want to use 3d space, so we use:
glVertex3f
Which takes the parameters of:
x
y
z
In that order.

So if we go to our code, we have:
glBegin(GL_QUADS);

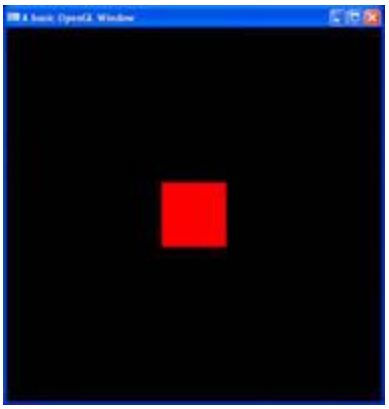
Now to set up our vertices I am simply calling:

```
glBegin(GL_QUADS); //begin the four sided shape
glVertex3f(-0.5, -0.5, 0.0); //first corner at -0.5, -0.5
glVertex3f(-0.5, 0.5, 0.0); //second corner at -0.5, 0.5
glVertex3f(0.5, 0.5, 0.0); //third corner at 0.5, 0.5
glVertex3f(0.5, -0.5, 0.0); //fourth corner at 0.5, -0.5
```

Then after we are done calling our vertices, we need to specify the end of our shape. To do this well call:
glEnd();

And there we have it, a nice little cube.

Coloring Shapes in OpenGL



Now that we are able to display objects on the screen. They don't look too flash if everything is the same colour. So we want to be able to change the colour of our current objects.

Now this is easy enough. There is a function you can call that looks like:
glColor3f(1.0, 0.0, 0.0);
This allows you to set the colour of the object to be drawn to the values of R(ed),G(reen),B(lue) where I have R set to 1, which is the maximum, G is set to 0 and B is also set to 0.

So here the highest value you can have for a colour is 1, and the lowest is 0. If you were to vary it to:
glColor3f(0.0, 1.0, 0.0);
We now have no red, full green, and no blue. So this would create a green object.

The colours can also be varied, for example:
glColor3f(0.0, 0.0, 0.0);
Wouls come out as black as there is no colour, and
glColor3f(1.0, 1.0, 1.0);
Would come out as white as it is using each colour fully.

So here "**glColor3f(1.0, 0.0, 0.0);**" will come out as a red object.

There is also another function that takes four parameters (RGBA) but I will show you that in the Blending tutorial. As it involves a completely new method.

Just keep in mind that these colours will not work while you have lighting effects enabled unless you call:
glEnable(GL_COLOR_MATERIAL);

Creating an OpenGL Cube using GLUT

In glut we are given a group of different 3D shapes to play with
one of them is the cube, it can be drawn with the line:
glutWireCube(2);
or:
glutSolidCube(2);

The 2 is the length of the sides, as it is a cube, all sides are the same.
some other 3D shapes to play with are the:

(for these 3, the more slices and stacks, the smoother the surface.)
glutSolidCone (base length, height, slices, stacks);
glutSolidSphere (radius, slices, stacks);
glutSolidTorus (inner radius, outter radius, slices, stacks);

alternatively there is also the Wire version which is done by replacing
the word Solid with Wire.

There is also a little teapot when playing with different effects, so that
you can see how it would change a more 'modelled' shape which can be called with

glutSolidTeapot(size);

How to rotate a shape in OpenGL

ust to make sure you know, OpenGL uses vectors.
So when using vectors, you can scale, translate and finally rotate.

To do a rotation in OpenGL, you first need to know the angle in which you wish to rotate the current object.
This can just be held as a number which I have done with:
int angle = 0;

If you scroll down to the Display function, you will see:
glClear (GL_COLOR_BUFFER_BIT);
glLoadIdentity();
gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glRotatef(angle, 1.0, 0.0, 0.0); //rotate on the x axis
glRotatef(angle, 0.0, 1.0, 0.0); //rotate on the y axis
glRotatef(angle, 0.0, 0.0, 1.0); //rotate on the z axis
glColor3f(1.0, 0.0, 0.0);
glutWireCube(2);
glFlush();
angle += 1;

Just forget the first 3 lines as we have looked at these, and skip straight to:
glRotatef(angle, 1.0, 0.0, 0.0); //rotate on the x axis
glRotatef(angle, 0.0, 1.0, 0.0); //rotate on the y axis
glRotatef(angle, 0.0, 0.0, 1.0); //rotate on the z axis

The rotate function takes 4 parameters, these are:
angle of rotation
whether or not to rotate around the x axis
whether or not to rotate around the y axis
whether or not to rotate around the z axis

In that order.

So when selecting the axis to rotate around, 1 is yes, and 0 is no.

You will see that I am using the variable angle, this is set to 0
at the beginning, and if you look at the last line in Display:
angle += 1;
You can see that I am increasing the angle by 1(thats 1 degree) every
time OpenGL goes through the Display function.

Therefore:
glRotatef(angle, 1.0, 0.0, 0.0); //rotate on the x axis
glRotatef(angle, 0.0, 1.0, 0.0); //rotate on the y axis
glRotatef(angle, 0.0, 0.0, 1.0); //rotate on the z axis
Translates to:
Rotate angle degrees on the x,y and z axis

Because we have the angle updating in the display function
we call:
glutIdleFunc (display);
in the Main function, underneath the glutDisplayFunc command.

NOTE: the cube will look like a giant mess, check the smooth
rotation tutorial on how to fix this.

Smooth Rotation in OpenGL using Double Buffering

Last time we looked at rotation the cube looked 'broken' now it looked like this because we had a single buffer running. This means that the program is automatically drawing straight to the window. To fix this we add a second buffer, by changing the:
glutInitDisplayMode (GLUT_SINGLE);
to:
glutInitDisplayMode (GLUT_DOUBLE);
this is giving the programa buffer to draw what it has to, then transfer what is acctually needed to the screen.
You may notice that alot of games these days even have a triple buffer, this is pretty redundant in OpenGL, in fact it doesn't even exist in OpenGL.

When changing the buffer from Single to Double we also have to tell the program to swap the buffers, so we acctually see what is on the second, and not just what we saw before.

To do this, change the line in the 'display' function that says:
glFlush();
to:
glutSwapBuffers();

And your done, the cube should be rotating perfectly.