UNIT 10

INTRODUCTION TO OPENGL

Introduction

OpenGL is the industry's most widely used environment for the development of portable, interactive 2D and 3D graphics applications. The OpenGL API is open source and is platform independent. 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). openGL is portable

windows, Mac (various platforms)

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. 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.

openal is a software interface to graphics hardware. The interface consists of over 250 different function calls which can be used to draw complex two and three dimensional scenes from simple geometric primitives such as points, lines & polygons. It is cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is used to interact with a graphics processing unit (GPU), to achieve hardware accelerated rendering.

Why OpenGL?

The first vendor-independent API for development of graphics applications

There is no need to license it to use it.

It was designed to use the graphics card where possible to improve performance.

Originally based on a state machine, procedural model; thus it can be used with a wide variety of programming languages.

Access to Libraries

Although the OpenGL interface provides platform independence, the set-up of libraries and header files varies across systems and compilers and we cannot hope to cover all such variations. Typically the initial header inclusions for a Windows system will include

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

Plus any other library access (maths.h etc) that we may require. This will be slightly different on Linux where if we are using X windows then a likely include will be

```
#include <X11/Xlib.h>
#include <GL/ glut.h>
```

Callback functions

Callbacks simplify event processing for the application developer. Callbacks simplify the Process by defining what actions are supported, and automatically handling the user events. A Callback function is a function which the library (GLUT) calls when it needs to know how to process something. E.g. when GLUT gets a key down event, it uses the glutKeyboardFunc callback routine to find out what to do with a key press.

The GLU callback functions, gluBeginPolygon, gluTessVertex, gluNextContour, and gluEndPolygon, are similar to the OpenGL polygon functions. They typically save the data for the triangles, triangle meshes, and triangle fans in user-defined data structures or in OpenGL display lists. To render the polygons, other code traverses the data structures or calls the display lists. Although the callback functions could call OpenGL functions to display polygons directly, this is usually not done, as tessellation can be computationally resource-intensive. It s a good idea to save the data if there is any chance that you want to display it again. The GLU tessellation functions are guaranteed never to return any new vertices, so interpolation of vertices, texture coordinates, or colors is never required.

The following are valid callback functions:

Callback	Description
GLU_TESS_ BEGIN	The GLU_TESS_BEGIN callback is invoked like glBegin to indicate the start of a (triangle) primitive. The function takes a single argument of type GLenum. If you set the GLU_TESS_BOUNDARY_ONLY property to GL_FALSE, the argument is set to either GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES. If you set the GLU_TESS_BOUNDARY_ONLY property to GL_TRUE, the argument is set to GL_LINE_LOOP. The function prototype for this callback is as follows: void begin(GLenum type);
GLU_TESS_ BEGIN_DAT	GLU_TESS_BEGIN_DATA is the same as the GLU_TESS_BEGIN callback except that it takes an additional pointer argument. This pointer is identical A to the opaque pointer provided when you call gluTessBeginPolygon. The function prototype for this callback is: void beginData (GLenum type, void * polygon_data);
GLU_TESS_ EDGE_FLAG	The GLU_TESS_EDGE_FLAG callback is similar to glEdgeFlag. The function takes a single Boolean flag that indicates which edges lay on the polygon boundary. If the flag is GL_TRUE, then each' vertex that follows begins an edge that lies on the polygon boundary; that is, an edge which separates an interior region from an exterior one. If the flag is GL_FALSE, then each vertex that follows begins an edge that lies in the polygon interior. The GLU_TESS_EDGE_FLAG callback (if defined) is invoked before the first vertex callback is made. Because triangle fans and triangle strips do not support edge flags, the begin callback is not called with GL_TRIANGLE_FAN or GL_TRIANGLE_STRIP if an edge flag callback is provided. Instead, the fans and strips are converted to independent triangles. The function prototype for this callback is: void edgeFlag (GLboolean flag);
GLU_TESS_ EDGE FLAG	The GLU_TESS_EDGE_FLAG_DATA callback is the same as the GLU_TESS_EDGE_FLAG callback except that it takes an additional pointer _DATA argument. This pointer is identical to the opaque pointer provided when you call gluTessBeginPolygon. The function prototype for this callback is: void edgeFlagData (GLboolean flag, void * polygon_data);

Drawing Primitives

We have used a limited set of drawing primitives and we will now begin to generalize capabilities. Drawing is accomplished using points, lines, polylines and polygons all of which are defined by vertices. OpenGL provides beyond the primitives of points (GL_POINTS) to include a segmented lines (GL_LINES) facility, to have joined lines (GL_LINE_STRIP) and closed loops (GL_LINE_LOOP). Further surfaces may be patched using polygons, triangles and quadrilaterals. A set of primitives required to render the image occur between the lines glBegin() to glEnd(). glBegin() takes an argument glvalue that governs how the vertex list is to be processed.

glBegin(Glvalue): start of vertex list where argument glvalue specifies the geometric primitive to be constructed at vertices.

Value	Meaning
GL__POINTS	individual points at vertices
GL_LINES	individual line segments / pair of vertices
GL_LINE_STRIP	connected line segments / series of connected line segments
GL__LINE_LOOP	connected line segments including first and last
GL_TRIANGLES	vertex triples interpreted as triangles
GL_TRIANGLES_STRIP	linked strip of triangles / Triple of vertices interpreted as traingle
GL_TRIANGLES_FAN	linked fan of triangles
GL__QUADS	vertex quadruples interpreted as 4 sided polygons
GL_QUADS_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of simple, convex polygon
glEnd()	

Drawings pixels, lines, polygons using OpenGL

All geometric primitives are eventually described in terms of their vertices coordinates that define the points themselves, the endpoints of line segments, or the corners of polygons. In any OpenGL implementation, floating-point calculations are of finite precision, and they have round-off errors. Consequently, the coordinates of OpenGL points, lines and polygons suffer from the same problems. Another more important difference arises from the limitations of a raster graphics display. On such a display, the smallest displayable unit is a pixel, and although pixels might be less than 1/ 100 of an inch wide, they are still much larger than t mathematician's concepts of infinitely small

(for points) or infinitely thin (for lines). When OpenGL performs calculations, it assumes points are represented as vectors of floating point numbers. However, a point is typically drawn as a single pixel, and many different points with slightly different coordinates could be drawn by OpenGL on the same pixel.

Points

A point is represented by a set of floating-point numbers called a vertex. All internal calculations are done as if vertices are three-dimensional. Vertices specified by the user as two dimensional (that is, with only x and y coordinates) are assigned a z coordinate equal to zero by OpenGL.

OpenGL works in the homogeneous coordinates of three-dimensional projective geometry, so for internal calculations, all vertices are represented with four floating-point coordinates (x, y, z, w). If w is different from zero, these coordinates correspond to the Euclidean three\ dimensional point (x/w,y/w, z/w). To control the size of a rendered point, use glPointSize() and supply the desired size in pixels as the argument.

void glPointSize(GLfloat size);

Sets the width in pixels for rendered points; size must be greater than 0.0 and by default is 1.0.

Lines

In OpenGL, the term line refers to a line segment, not the mathematician's version that extends to infinity in both directions. There are easy ways to specify a connected series of line segments, or even a closed, connected series of segments. In all cases, though, the lines constituting the connected series are specified in terms of the vertices at their endpoints.

With OpenGL, we can specify lines with different widths and lines that are stippled in various ways-dotted, dashed, drawn with alternating dots and dashes, and so on.

Drawing a line with OpenGL is apparently quite simple using the following code

```
glBegin(GL_LINES); — Marks the beginning of a vertex-data list that describes a geometric
glVertex2i(x0, y0);                                                    primitive line
glVertex2i(x1, y1);
glEnd()
```

The sample below shows how to draw 2D points and lines in openGL. We can consider it as the 'Hello World' program of openGL. This is the program of openGL in c/c++.

```
#include <GL/glut.h>
void init2D(float 1', float 3, float b)
{
glClearColor(r,g,b,0.0);
glMatrixMode (GL_PROJECTION);
gluOrthoZD (0.0, 200.0, 0.0, 150.0);
}
void display(Void)
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_POINTS);/ / draw two points
for(inti = 0; i < 10; i++)
{
glVertex2i(10+5*i,110);
}
glEnd();
glBegin(GL_LINES);// draw a line
glVertex2i(10,10);
glVertex2i(100,100);
glEnd();
glFlush();
}
void main(int argochar *argVU)
{
glutInit(&argc,argV);
glutIm'tDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow ("points and lines");
init2D(0.0,0.0,0.0);
glutDisplayFunc(display);
glutMainLoop();
}
```

Polygon

Polygons are typically drawn by filling in all the pixels enclosed within the boundary, but you can also draw them as outlined polygons or simply as points at the vertices. A filled polygon might be solidly filled or stippled with a certain

pattern; Although the exact details are omitted here, fined polygons are drawn in such a way that if adjacent polygons share an edge or vertex, the pixels making up the edge or vertex are drawn exactly once-they're included in only one of the polygons. This is done so that partially transparent polygons don't have their edges drawn twice, which would make those edges appear darker (or brighter, depending on what color you're drawing with).
A polygon has two sides-front and back-and might be rendered differently depending on which side is facing the viewer. This allows you to have cutaway Views 0f solid Object in which there is an obvious distinction between the parts that are inside and those that are outside. By default, both front and back faces are drawn in the same way. To Change this, or to draw only outlines or vertices, use
glPolygonMode()
We can draw polygon by following code of segment;

```
glBegin(GL_POLYGON); // Draw A Quad
glVertex3f(-0.5f, 1.0f, 0.0f); / / Top Left
glVertex3f(-1.0f, 0.0f, 0.0f); / / Left
g1Vertex3f(-O.5f, 1.0f, 0.0f); / / Bottom Left
glVertex3f(0.5f, 1.0f, 0.0f); / / Top Right
glVertex3f(1.0f, 0.0f, 0.0f); / / Right
glVertex3f(0.5f, 1.0f, 0.0f); / / Bottom Right
glEnd();
```

Complete example of drawing a polygon in OpenGL by using C/ C++ as below:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <glut.h>
void myInit(void)
{
glClearColor(1.0,1.0,1.0,0.0);
glColor3f(1.0f,0.0f,0.0f);
glPointSize(4.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0,800.0,0.0,600.0);
}
void myDisplay(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_POLYGON);
glVertex2i(100,100);
glVertex2i(100,300);
glVertex2i(400,300);
glVertex2i(600,150);
glVertex2i(400,100);
glEnd();
glFlush();
}
int main(int argochar * argvu)
{
glutInit(&argC,argv);
glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
glutInitWindOWSize(800,600);
glutInitWindowPosition(100,100);
glutCreateWindow("opengl Window");
glutDisplayFunc(myDisplay);
myInit();
glutMainLoop();
getchQ;
return 0; }
```

Colormap means a color table or a palette as an array of colors used to map pixel data.

Color commands

OpenGL has two color modes: the RGBA mode and color index mode. In RGBA mode, a color is specified by three intensities (for the Red, Green, and Blue components of the color) and optionally a fourth value, Alpha, which controls transparency. The function

glColor4f(red, green, blue, alpha)

Maps available red, green, and blue intensities onto (0.0, 1.0) where 0.0 means that the Color component is absent ((0.0, 0.0, 0.0) is black) and 1.0 is a saturated color ((1.0, 1.0, 1.0) is White). The number of bits used to represent each color depends upon the graphics card. Current graphics cards have several Mbytes of memory, and use 24 or 32 bits for color (24-bit: 8 bits per color; 32-bit: 8 bits per color + 8 bits padding or transparency). The term bit plane refers to an image of single-bit values. Thus, a system with 24 bits of color has 24 bit planes.

For a system with 8 bit planes, 28 = 256 different colors could be displayed simultaneously These 256 colors could be selected from a set of 2m colors. For m = 24 this gives 22 4≈16 million colors. The color table could be altered to give different palettes of colors. OpenGL supports this mode, called color index mode.

In index-color mode the number of simultaneously available colors is limited by the size of color map and the number of bitplanes available. The size of color map is determined by the amount of hardware dedicated to it

With RGBA mode, each pixel's color is independent of other pixel. However in color indexed mode each pixel with the same index stored in its bitplanes shares the same color-map location.

Using glColor3f

glColor3f() takes 3 arguments: the red, green and blue components of the color. After we use glColor3f, everything draw will be in that color. For example, consider this display function:

```
void display()
{
glC1ear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glColor3f(0.5f, 0.0f, 1.0f); // (0.5, 0, 1) is half red and full blue, giving purple.
glBegin(GL_QUADS);
glVertex2f(-0.75, 0.75);
glVertex2f(-0.75, -0.75);
glVertex2f(0.75, -0.75);
glVertex2f(0.75, 0.75);
glEnd();
glutSwapBuffers();
}
```

[If the contents of an entry in the color map change, then all pixels of that color index change their color]

Giving Individual Vertices Different Colors

glColor3f can be called in between glBegin and glEnd. When it is used this way, it can be used to give each vertex its own color. The resulting rectangle is then shaded with an attractive color gradient.

```
void display()
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glBegin(GL_QUADS);
glColor3f(1.0f, 1.0f, 1.0f); // make this vertex purple
glVertex2f(-0.75, 0,75);
glColor3f(1.0f, 1.0f, 1.00; // make this vertex red
glVertex2f(-O.75, -0.75);
glColor3f(1.0f, 1.0f, 1.00; / / make this vertex green
glVertex2f(0.75, -0.75);
glColor3f(1.0f, 1.0f, 1.0f); // make this vertex yellow
glVertex2f(0.75, 0.75);
glEnd();
glutSwapBuffers();
}
```

OpenGl Color Codes

For Background color:

gl.glClearColor(float red, float green, float blue, float alpha)

```
glClearColor(0.0,0.0,1.0,0.0);/ / dark blue
```

Fill color:

```
glColor4f(1.0f, 0.0f, 0.0f, 0.0f);/ /red
glColor4f(1.0f, 1.0f, 1.0f, 0.0f);/ /white
glColor4f(1.0f, 1.0f, 0.0f, 0.0f);/ /yellow
glColor4f(1.0f, 0.0f, 1.0f, 0.0f);/ /purple
```

```
glColor4f(0.0f, 1.0f, 1.0f, 1.0f);/ / light blue
glColor4f(1.0f, 0.5f, 0.0f, 0.0f);/ / orange/ brown
glColor3f(0.0f, 1.0f, 0.0f);/ /Green
glColor3f(0.0f, 0.0f, 1.0f);/ / Blue
glColor3f(0.5f, 1.0f, 1.0f);/ /cyan
glColor3f(0.0f, 0.0f, 0.0f);/ / Black
glColor3f(1.0f, 0.0f, 1.0f);/ /Purple
glColor3f(1.0f, 0.5f, 0.0f);/ / Orange
glColor3f(0.5f, 0.5f, 0.5f);/ / Violet
glColor3f(0.0f, 0.5f, 0.5f);/ / Blue-Green
glColor3f(0.0f, 0.5f, 1.0f);/ /baby Blue
glColor3f(2.0f, 0.5f, 1.0f);/ / Lilac
glColor3f(0.1f, 0.1f, 0.1f);/ / Dark grey
glColor3f(0.1f, 0.0f, 0.1f);/ / Dark Purple
glColor3f(O.1f, 0.1f, 0.0f);/ /Bronze
glColor3f(0.0f, 0.1f, 0.1f);/ /Dark blue
glColor3f(0.0f, 0.1f, 0.0f);/ /Forest Green
glColor3f(0.1f, 0.0f, 0.0f);/ / Brown
```

Basic Lighting

Lighting in the real world is extremely complicated and depends on way too many factors, something we can't afford to calculate on the limited processing power we have. Lighting in OpenGL is therefore based on approximations of reality using simplified models that are much easier to process and look relatively similar. These lighting models are based on the physics of light as we understand it. One of those models is called the Phong lighting model. The building blocks of the Phong model consist of 3 components: ambient, diffuse and Specular lighting.

Ambient lighting: even when it is dark there IS usually still some light somewhere in the world (the moon, a distant light) so objects are almost never completely dark. To simulate this we use an ambient lighting constant that always gives the object some color. Ambient illumination is light that has been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has first bounced off many surfaces. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, its scattered equally in all directions.

Diffuse lighting: simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes. The diffuse component is the light that comes from one direction, so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. Any light coming from a particular position or direction probably has a diffuse component.

Specular lighting: it simulates the bright spot of a light that appears on shiny objects. Specular highlights are often more inclined to the color of the light than the color of the object. Specular light comes from a particular direction, and it tends to bounce off the surface in a preferred direction. A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high Specular component, and chalk or carpet has almost none. You can think of secularity as shininess.