Introduction to OpenGL: Part 1

OpenGL is concerned with the rendering of a scene onto a framebuffer. A framebuffer is the region of memory that is copied to the screen for display.

OpenGL renders primitives, subject to a set of modes. Primitives are points, line segments, polygons, or pixel rectangles.

Primitives are specified, modes are set, and other OpenGL operations are accomplished by sending commands in the form of procedure calls or functions.

There are OpenGL bindings to a number of languages; we will look at the C language bindings.

Primitives are defined by a set of one or more vertices. Vertices define points, which may be endpoints of edges, corners of a polygon, or simply points, in two or more dimensions.

Attributes other than position (e.g., color, texture) may also be be assigned to a vertex.

The model for interpreting OpenGL is the client-server model (more on this later).

OpenGL command syntax

OpenGL commands are functions or procedures, and follow a simple naming convention. The following shows the `glVertex` command:

`void glVertexnt (args)`

where
`n` is the number of dimensions (2,3, or 4) and
`t` indicates the data type (integer, float, double, etc.).

For example, we could have

```
glVertex2f(x1, y1);    /* 2 dimension, type float  */
glVertex2i(i1, j1);    /* 2 dimension, type int    */
```

Another form of the `glVertex` command can use a pointer to an array of vertices, as follows:

```
GLfloat vertex[3]      /* an array of 3 elements     */
glVertex3fv(vertex);   /* 3 D, type pointer to float */
```

The `v` in `glVertex3fv` indicates an array (i.e. a vector) of vertices.

Sets of vertices are used to specify points, lines, polygons, and surfaces using a `glBegin - glEnd` construct; e.g., for a line:

```
glBegin(GL_LINES);
    glVertex2f(x1, y1);
    glVertex2f(x2, y2);
glEnd();
```

We can define a set of points similarly:

```
glBegin(GL_POINTS);
    glVertex2f(x1, y1);
    glVertex2f(x2, y2);
    glVertex2f(x3, y3);
glEnd();
```

In general, objects are defined with primitives of the form:

```
glBegin(object_type);
    glVertex..(...);
         ...
    glVertex..(...);
glEnd();
```

Other `object_types` are polylines (lines made up of several, connected, line segments):

```
GL_LINE_STRIP
```

The closed version is

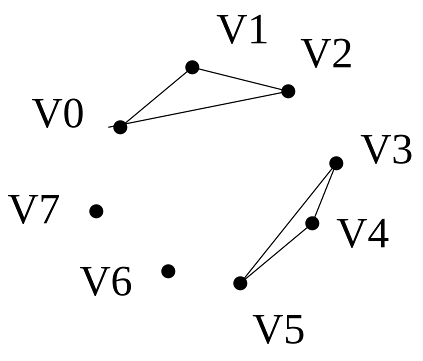
```
GL_LINE_LOOP
```

For solid surfaces, the `object_types` are polygons, triangles, quadrilaterals, and special types called strips and fans.

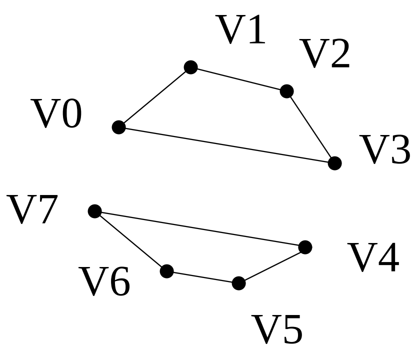
```
GL_POLYGON
GL_TRIANGLES
GL_QUADS
GL_TRIANGLE_STRIP
GL_QUAD_STRIP
```

The tutorial `shapes` shows the use of the primitive drawing functions.

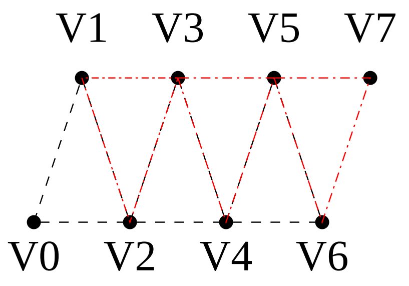
Note that OpenGL only guarantees that convex polygons with non-intersecting edges will be rendered correctly.



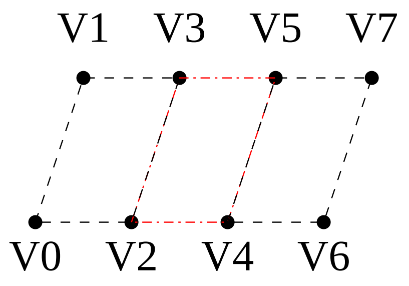
GL_TRIANGLES



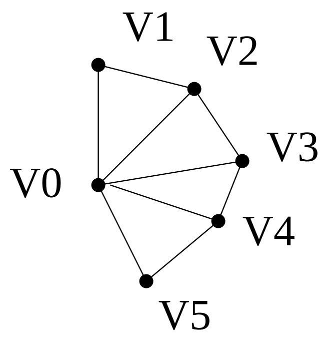
GL_QUADS



GL_TRIANGLE_STRIP



GL_QUAD_STRIP



GL TRIANGLE FAN

Other Attributes – color

Although color can be represented in a number of different ways, we will initially use the RGB (red - green - blue) color model. In fact, we will extend this to the RGBA model, where A (alpha) is a measure of the transparency.

The quadruple

`(1.0, 1.0, 1.0, 0.0)`

would represent solid, opaque white — opaque because the transparency is 0.0, and white because all of red, green, and blue are saturated.

`(1.0, 0.0, 0.0, 0.0)`

would represent solid, opaque red — only the red component is present.

what would the following represent?

`(0.0, 0.0, 0.0, 0.0)`

`(0.0, 1.0, 0.0, 0.0)`

`(0.0, 0.0, 0.5, 0.0)`

The tutorial **`shapes`** shows the use of both the primitive drawing functions and color.

Projections

We are interested in a 2-dimensional display of things from a 3-dimensional world. We must project the 3-dimensional objects onto the plane.

One of the simplest projections is the orthographic projection. Here, in effect, the viewer looks directly along one of the axes of the coordinate system — usually the z-axis. (Much more on projections to come.)

Associated with the projection is the viewport. Primitives that project within the viewport are clipped, while primitives that project entirely outside the viewport are not displayed at all.

The projection of these primitives onto the viewing plane is something we will discuss in detail later. For the present, the function

```
glOrtho(left, right, bottom, top, near, far)
```

defines a viewing volume bounded by (`left,right`) in the x-dimension, (`bottom,top`) in the y-dimension, and (`near,far`) in the z-dimension.

The display environment

Although OpenGL is often used for event driven, interactive applications, OpenGL itself does not have functions for user interaction or for interacting with the local environment (creating a display window, for example).

These functions are performed by another library (in our case, the `GLUT` library) which sets up the display environment — window size, position, etc. and, in fact, calls the OpenGL functions we will write.

We will discuss the `GLUT` library in more detail later. Initially, we will use “boiler plate” functions into which we will embed our code.

The following simple OpenGL program draws a black rectangle on a white background.

A simple program — `hello.c`

```
/* hello.c
 *
 * Copyright (c) 1993-1997, Silicon Graphics, Inc.
 * ALL RIGHTS RESERVED
 *
 * This is a simple, introductory OpenGL program.
 * It draws a black square on a white background.
 */
#include <GL/glut.h>

void init (void)
/* this function sets the initial state */
{
/* select clearing (background) color  to white  */
   glClearColor (1.0, 1.0, 1.0, 0.0);

/* initialize viewing values  */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
```

```
void display(void)

{
/* clear all pixels  */
   glClear (GL_COLOR_BUFFER_BIT);


/* draw black polygon (rectangle) with corners at
 * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 */
   glColor3f (0.0, 0.0, 0.0);
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();


/* don't wait!
 * start processing buffered OpenGL routines
 */
   glFlush ();
}
```

```
int main(int argc, char** argv)
/*
 * Declare initial window size, position, and
 * display mode (single buffer and RGBA).
 * Open window with "hello" * in its title bar.
 * Call initialization routines.
 * Register callback function to display graphics.
 * Enter main loop and process events.
 */
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("hello");

   init ();

   glutDisplayFunc(display);

   glutMainLoop();
   return 0;   /* ANSI C requires main to return int. */
}
```

Note that this does not look much like a conventional program — there is really no "flow of control" to sequence events. This is characteristic of event driven code — functions are written which "handle" the various events. In this case the event handlers are callback functions.

In this simple example, there are no external events to handle; there is no user interaction required, and any other types of interaction (e.g. resizing a window) assume the system default case.

Two functions were defined:

`init()` is simply a convenient place to put the initialization information (the background color and projection)

`display()` is the function defining what is to be displayed in the graphics window.

The functions `glutInitWindowPosition(), glutInitWindowSize()`, and `glutCreateWindow()` set up a display window in a particular place, with a particular size and window title.

`glutDisplayFunc()` sets the function to be called to perform the actual drawing (`display()` in the present case).

`glutMainLoop()` enters the GLUT event processing loop (and never returns). Registered callback functions will be called as events occur.