CA417 - Computer Graphics

OpenGL - Part 1

David Sinclair

Overview

OpenGL is an Application Programmer's Interface (API) that provides a "medium to high level" interface to the graphics hardware. It is sufficiently abstract to isolate you from the hardware details, while being close enough to the hardware to run efficiently.

Overview

OpenGL is an Application Programmer's Interface (API) that provides a "medium to high level" interface to the graphics hardware. It is sufficiently abstract to isolate you from the hardware details, while being close enough to the hardware to run efficiently.

There are 3 different ways to view the OpenGL library.

- The Programmer's View

Overview

OpenGL is an Application Programmer's Interface (API) that provides a "medium to high level" interface to the graphics hardware. It is sufficiently abstract to isolate you from the hardware details, while being close enough to the hardware to run efficiently.

There are 3 different ways to view the OpenGL library.

- The Programmer's View
- The OpenGL State Machine

Overview

OpenGL is an Application Programmer's Interface (API) that provides a "medium to high level" interface to the graphics hardware. It is sufficiently abstract to isolate you from the hardware details, while being close enough to the hardware to run efficiently.

There are 3 different ways to view the OpenGL library.

- The Programmer's View
- The OpenGL State Machine
- The OpenGL Pipeline

The Programmer's View

From the programmer's point of view we need to be able to:

- specify the objects in a scene;

The Programmer's View

From the programmer's point of view we need to be able to:

- specify the objects in a scene;
- describe their properties,

The Programmer's View

From the programmer's point of view we need to be able to:

- specify the objects in a scene;
- describe their properties, and;
- define how these objects should be viewed.

The Programmer's View

From the programmer's point of view we need to be able to:

- specify the objects in a scene;
- describe their properties, and;
- define how these objects should be viewed.

Objects are either geometric objects, such as points, lines, curves, polygons etc., or they may be images.

The Programmer's View

From the programmer's point of view we need to be able to:

- specify the objects in a scene;
- describe their properties, and;
- define how these objects should be viewed.

Objects are either geometric objects, such as points, lines, curves, polygons etc., or they may be images.

For efficiency purposes, most Graphics systems separate the shape of the object from how it is displayed. A dashed red line has a shape, a line, and a set of properties, i.e. it is read and it is dashed.

The Programmer's View

From the programmer's point of view we need to be able to:

- specify the objects in a scene;
- describe their properties, and;
- define how these objects should be viewed.

Objects are either geometric objects, such as points, lines, curves, polygons etc., or they may be images.

For efficiency purposes, most Graphics systems separate the shape of the object from how it is displayed. A dashed red line has a shape, a line, and a set of properties, i.e. it is read and it is dashed.

How an object is viewed depends on the positioning of the "camera", which we will call the viewpoint, and the illumination of the environment.

The OpenGL State Machine

We can also think of OpenGL as a **state machine** with inputs and outputs. The output of the state machine is a set of **pixels** that describe the image we see from the viewpoint. The inputs are provided by function calls that define:

- the objects (geometric and images);

The OpenGL State Machine

We can also think of OpenGL as a **state machine** with inputs and outputs. The output of the state machine is a set of **pixels** that describe the image we see from the viewpoint. The inputs are provided by function calls that define:

- the objects (geometric and images);and
- the current state of the OpenGL state machine.

The OpenGL State Machine

We can also think of OpenGL as a **state machine** with inputs and outputs. The output of the state machine is a set of **pixels** that describe the image we see from the viewpoint. The inputs are provided by function calls that define:

- the objects (geometric and images);and
- the current state of the OpenGL state machine.
    - colour
    - material properties
    - viewing conditions

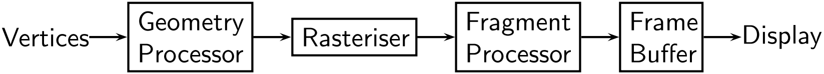
The OpenGL Pipeline

Sometimes it is useful to consider OpenGL from an implementation viewpoint. OpenGL, like most graphics hardware systems, is based on a **pipeline mode**.

Vertices → Geometry Processor → Rasteriser → Fragment Processor → Frame Buffer → Display

The OpenGL Pipeline

Sometimes it is useful to consider OpenGL from an implementation viewpoint. OpenGL, like most graphics hardware systems, is based on a **pipeline mode**.



- Geometric primitives are defined by the vertices (points in space).

The OpenGL Pipeline

Sometimes it is useful to consider OpenGL from an implementation viewpoint. OpenGL, like most graphics hardware systems, is based on a **pipeline mode**.

Vertices → Geometry Processor → Rasteriser → Fragment Processor → Frame Buffer → Display

- Geometric primitives are defined by the vertices (points in space).
- The Geometry Processor can transform (rotate, translate and scale), project and clip the geometric objects.

The OpenGL Pipeline

Sometimes it is useful to consider OpenGL from an implementation viewpoint. OpenGL, like most graphics hardware systems, is based on a **pipeline mode**.

Vertices → Geometry Processor → Rasteriser → Fragment Processor → Frame Buffer → Display

- Geometric primitives are defined by the vertices (points in space).
- The Geometry Processor can transform (rotate, translate and scale), project and clip the geometric objects.
- The Rasteriser generates **fragments** (potential pixels) that may lie inside the Frame Buffer.

The OpenGL Pipeline

Sometimes it is useful to consider OpenGL from an implementation viewpoint. OpenGL, like most graphics hardware systems, is based on a **pipeline mode**.

Vertices → Geometry Processor → Rasteriser → Fragment Processor → Frame Buffer → Display

- Geometric primitives are defined by the vertices (points in space).
- The Geometry Processor can transform (rotate, translate and scale), project and clip the geometric objects.
- The Rasteriser generates **fragments** (potential pixels) that may lie inside the Frame Buffer.
- The Fragment Processor determines which fragments are visible and colour them appropriately.

The OpenGL Pipeline

Sometimes it is useful to consider OpenGL from an implementation viewpoint. OpenGL, like most graphics hardware systems, is based on a **pipeline mode**.

Vertices → Geometry Processor → Rasteriser → Fragment Processor → Frame Buffer → Display

- Geometric primitives are defined by the vertices (points in space).
- The Geometry Processor can transform (rotate, translate and scale), project and clip the geometric objects.
- The Rasteriser generates **fragments** (potential pixels) that may lie inside the Frame Buffer.
- The Fragment Processor determines which fragments are visible and colour them appropriately.
- The Frame Buffer contains the pixels to be displayed on the viewing device.

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

Primitive These specify elements of the image, both geometric (in 2D and 3D) and discrete (bitmap images).

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

Primitive These specify elements of the image, both geometric (in 2D and 3D) and discrete (bitmap images).

Attribute These control the appearance of primitives.

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

Primitive These specify elements of the image, both geometric (in 2D and 3D) and discrete (bitmap images).

Attribute These control the appearance of primitives.

Viewing These control the position and orientation of the viewpoint, as well as the lens attached to the camera at the viewpoint.

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

Primitive These specify elements of the image, both geometric (in 2D and 3D) and discrete (bitmap images).

Attribute These control the appearance of primitives.

Viewing These control the position and orientation of the viewpoint, as well as the lens attached to the camera at the viewpoint.

Control These enable or disable OPenGl features (such as lighting, texture mapping and hidden-surface removal).

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

Primitive These specify elements of the image, both geometric (in 2D and 3D) and discrete (bitmap images).

Attribute These control the appearance of primitives.

Viewing These control the position and orientation of the viewpoint, as well as the lens attached to the camera at the viewpoint.

Control These enable or disable OPenGl features (such as lighting, texture mapping and hidden-surface removal).

Query These allow us to query the values of OpenGL state values and the capabilities of particular systems.

Types of OpenGL functions

There are over 200 OpenGL functions. These can be grouped into:

- **Primitive** These specify elements of the image, both geometric (in 2D and 3D) and discrete (bitmap images).
- **Attribute** These control the appearance of primitives.
- **Viewing** These control the position and orientation of the viewpoint, as well as the lens attached to the camera at the viewpoint.
- **Control** These enable or disable OPenGl features (such as lighting, texture mapping and hidden-surface removal).
- **Query** These allow us to query the values of OpenGL state values and the capabilities of particular systems.
- **Input/Window** Strictly, these are not part of the core OpenGL libraries. They are provided in the GLUT library.

OpenGL Programming Conventions

The core OpenGL functions are contained in the `gl` library and the OpenGL Utility functions are contained in the `glu` library. Functions from the core library begin with `gl`, such as `glVertex3`, whereas functions from the utility library begin with `glu`, such as `gluOrtho2D`.

OpenGL Programming Conventions

The core OpenGL functions are contained in the `gl` library and the OpenGL Utility functions are contained in the `glu` library. Functions from the core library begin with `gl`, such as `glVertex3`, whereas functions from the utility library begin with `glu`, such as `gluOrtho2D`.

Since OpenGL functions need to handle different types of data, each Open GL function comes in various forms. For example, `glVertex3f` specifies a vertex in 3 dimensions using `floats`, whereas `glVertex2i` specifies a vertex in 2 dimensions using `ints`.

OpenGL Programming Conventions

The core OpenGL functions are contained in the `gl` library and the OpenGL Utility functions are contained in the `glu` library. Functions from the core library begin with `gl`, such as `glVertex3`, whereas functions from the utility library begin with `glu`, such as `gluOrtho2D`.

Since OpenGL functions need to handle different types of data, each Open GL function comes in various forms. For example, `glVertex3f` specifies a vertex in 3 dimensions using `floats`, whereas `glVertex2i` specifies a vertex in 2 dimensions using `ints`.

OpenGL functions that end with a `v` allow the programmer to pass a parameter by a pointer.

```
GLfloat point[3];
glVertex3fv (point);
```

My First OpenGL Program

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

void display (void)
{
  glClear (GL_COLOR_BUFFER_BIT);

  glBegin (GL_POLYGON);
    glVertex2f (-0.5, -0.5);
    glVertex2f (-0.5, 0.5);
    glVertex2f (0.5, 0.5);
    glVertex2f (0.5, -0.5);
  glEnd ();

  glFlush ();
  return;
}

int main (int argc, char **argv)
{
  glutInit (&argc, argv);
  glutCreateWindow ("simple");
  glutDisplayFunc (display);
  glutMainLoop ();
}
```

My First OpenGL Program (2)

Line 1 includes the GLUT (OpenGL Utility Toolkit). This is a library of windowing functions common in nearly all modern operating systems. GLUT uses lower level libraries such as `glx` (X11), `wgl` (Windows) or `agl` (OSX).

My First OpenGL Program (2)

Line 1 includes the GLUT (OpenGL Utility Toolkit). This is a library of windowing functions common in nearly all modern operating systems. GLUT uses lower level libraries such as `glx` (X11), `wgl` (Windows) or `agl` (OSX).

Line 20 initialises GLUT and should be called before other OpenGL or GLUT functions. Line 21 creates a new window given current state variables. It returns an integer so that the window can be referred to in multi-window applications.

My First OpenGL Program (2)

Line 1 includes the GLUT (OpenGL Utility Toolkit). This is a library of windowing functions common in nearly all modern operating systems. GLUT uses lower level libraries such as `glx` (X11), `wgl` (Windows) or `agl` (OSX).

Line 20 initialises GLUT and should be called before other OpenGL or GLUT functions. Line 21 creates a new window given current state variables. It returns an integer so that the window can be referred to in multi-window applications.

OpenGL and GLUT use **event loops** and **callbacks**. In this paradigm, common in interactive and real-time applications, the application responds to events. Events, such as mouse movement, mouse clicks, key presses, window movements and resizing, are placed in an **event queue** and are handled by the **event loop**. The **event loop** uses **callback functions** to handle each event.

My First OpenGL Program (3)

Our program uses only one **callback function**, the **display callback** (lines 3-16). Callback functions need to be registered with the OpenGL system. In line 22, the program registers the `display` function as the **display callback**. Whenever OpenGL determines that the screen needs to be redrawn, the **display callback** is invoked.

My First OpenGL Program (3)

Our program uses only one **callback function**, the **display callback** (lines 3-16). Callback functions need to be registered with the OpenGL system. In line 22, the program registers the `display` function as the **display callback**. Whenever OpenGL determines that the screen needs to be redrawn, the **display callback** is invoked.

The **event loop** is entered by the call to `glutMainLoop` in line 23. This function is never returned from unless the program is killed.

Code included after the call to `glutMainLoop` will never get executed. Some compilers expect `main` to return a value, so in those cases you can include the line

```
return (0);
```

after the call to `glutMainLoop`.

My First OpenGL Program (4)

Lines 7-12 draw the rectangle. They use function prototypes and constants that are contained in `GL/gl.h` and `GL/glu.h`. `GLUT/glut.h` (line 1) includes these files. `glBegin` specifies the start of a list of vertices that define an object. `glEnd` specifies the end of the list of vertices.

My First OpenGL Program (4)

Lines 7-12 draw the rectangle. They use function prototypes and constants that are contained in `GL/gl.h` and `GL/glu.h`. `GLUT/glut.h` (line 1) includes these files. `glBegin` specifies the start of a list of vertices that define an object. `glEnd` specifies the end of the list of vertices.

`glClear` (line 5) clears all the bits in the specified buffers. Each buffer has a specific identifier bit and multiple buffers can be cleared by or-ing the bits together. `glFlush` (line 14) forces all previous buffered OpenGL commands to execute.

Compiling OpenGL Programs

For UNIX systems, use

`cc myapp.c -o myapp -lgl - lglu - -lm -lX11`
(Sometimes you may need to add `-lXmu`.)

Compiling OpenGL Programs

For UNIX systems, use

`cc myapp.c -o myapp -lgl - lglu - -lm -lX11`
(Sometimes you may need to add `-lXmu`.)

In a Windows environment, `OpenGL32.dll` and `glu32.dll` should the in the system directories. The lib files should be in `..\VC\lib` and the include files should be in `..\VC\include\GL`. If you need to download the GLUT files (`glut32.dll, glut32.lib, glut.h`) from the web they go into the correspondding OpenGL directories.

Compiling OpenGL Programs

For UNIX systems, use

`cc myapp.c -o myapp -lgl - lglu - -lm -lX11`
(Sometimes you may need to add `-lXmu`.)

In a Windows environment, `OpenGL32.dll` and `glu32.dll` should the in the system directories. The lib files should be in `..\VC\lib` and the include files should be in `..\VC\include\GL`. If you need to download the GLUT files (`glut32.dll, glut32.lib, glut.h`) from the web they go into the correspondding OpenGL directories.

For OSX, the `glut.h` file is included by

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

and the cc command needs
`-framework OpenGL -framework GLUT`.

Colour in OpenGL

In OpenGL there are 2 colour modes:

Colour in OpenGL

In OpenGL there are 2 colour modes:

RGB/RGBA Each colour is made from a combination of Red, Green and Blue values. The colour values range from 0.0 t o 1.0. The 4^{th} component, alpha, is the **opacity**. An alpha value of 1.0 means the colour is opaque and you cannot see "what is behind" the pixel, whereas an alpha value of 0.0 means the pixel is transparent.

Colour in OpenGL

In OpenGL there are 2 colour modes:

RGB/RGBA
: Each colour is made from a combination of Red, Green and Blue values. The colour values range from 0.0 t o 1.0. The 4^{th} component, alpha, is the **opacity**. An alpha value of 1.0 means the colour is opaque and you cannot see "what is behind" the pixel, whereas an alpha value of 0.0 means the pixel is transparent.

colour-index
: An old historical technique when memory was expensive. A colour is specified as an index into a table of red, green and blue values. The table is preloaded and usually has 256 entries. Nowadays this is very rarely used.

Colour in OpenGL

In OpenGL there are 2 colour modes:

RGB/RGBA
: Each colour is made from a combination of Red, Green and Blue values. The colour values range from 0.0 t o 1.0. The 4^{th} component, alpha, is the **opacity**. An alpha value of 1.0 means the colour is opaque and you cannot see "what is behind" the pixel, whereas an alpha value of 0.0 means the pixel is transparent.

colour-index
: An old historical technique when memory was expensive. A colour is specified as an index into a table of red, green and blue values. The table is preloaded and usually has 256 entries. Nowadays this is very rarely used.

We will use RGB(A).

Colour in OpenGL (2)

Colour is part of the OpenGL state. There are two colour active in the OPenGL state, the **current drawing colour** and the **current clear colour**. The **current clear colour** is an RGBA value and is set by:

```
void glClearColor (GLclampf r, GLclampf, g,
                   GLclampf b, GLclampf a)
```

The **current drawing colour** is set by either:

```
void glColor3f (type r, type g, type b)
void glColor3fv (type *colour)
```

for RGB, and for RGBA by either:

```
void glColor4f (type r, type g, type b, type a)
void glColor4fv (type *colour)
```

There are `b, i, d, ub, us, ui` versions of the above OpenGL functions.

GLUT Default Values

We can change the default values used by GLUT.

```
void glutInitDisplayMode (unsigned int mode)
```

where `mode` specifies the colour model (`GLUT_RGB` or `GLUT_INDEX`) and the whether single (`GLUT_SINGLE`) or double buffering (`GLUT_DOUBLE`) is used.

GLUT Default Values

We can change the default values used by GLUT.

`void glutInitDisplayMode (unsigned int mode)`

where `mode` specifies the colour model (`GLUT_RGB` or `GLUT_INDEX`) and the whether single (`GLUT_SINGLE`) or double buffering (`GLUT_DOUBLE`) is used.

The size of the GLUT window is set by:

`void glutInitWIndowSize (int width, int height)`

and the position is set by:

`void glutInitWindowPosition (int x, int y)`

GLUT Default Values

We can change the default values used by GLUT.

```
void glutInitDisplayMode (unsigned int mode)
```

where `mode` specifies the colour model (`GLUT_RGB` or `GLUT_INDEX`) and the whether single (`GLUT_SINGLE`) or double buffering (`GLUT_DOUBLE`) is used.

The size of the GLUT window is set by:

```
void glutInitWIndowSize (int width, int height)
```

and the position is set by:

```
void glutInitWindowPosition (int x, int y)
```

Unlike OpenGL where (0,0) is at the bottom left of the screen, in GLUT (0,0) is at the upper left of the screen.

2-D Viewing

In 2-D viewing, we assume the camera is perpendicular to the x-y plane and all we need to do is define the minimum and maximum x and y of the **viewing** (**clipping**) window.

```
void gluOrtho2D (GLdouble left, GLdouble right,
                 GLdouble bottom, GLdouble top)
```

2-D Viewing

In 2-D viewing, we assume the camera is perpendicular to the *x*-*y* plane and all we need to do is define the minimum and maximum *x* and *y* of the **viewing** (**clipping**) window.

```
void gluOrtho2D (GLdouble left, GLdouble right,
                 GLdouble bottom, GLdouble top)
```

We can setup multiple **viewports** on a screen.

```
void glViewport (GLint x, GLint y,
                 GLsizei w, GLsizei h)
```

specifies a viewport of width `w` pixels and height `h` pixels whose lower left corner is located at (`x`,`y`).

Viewing Transformations

So far we have seen 2 coordinate systems, **world coordinates** and **screen coordinates**. As part of the rendering process, OpenGL automatically converts from **world coordinates** to **screen coordinates**, but OpenGL needs 2 pieces of information:

- the size of the display window (as determined by `glutInitWindowSize`), and
- how much of the world to display (as determined by `gluOrtho2D`).

Viewing Transformations

So far we have seen 2 coordinate systems, **world coordinates** and **screen coordinates**. As part of the rendering process, OpenGL automatically converts from **world coordinates** to **screen coordinates**, but OpenGL needs 2 pieces of information:

- the size of the display window (as determined by `glutInitWindowSize`), and
- how much of the world to display (as determined by `gluOrtho2D`).

`gluOrtho2D` sets up the **projection matrix** for 2 dimensions. The typical process for specifying a matrix is:

- Specify the matrix.
- Set the matrix to the identity matrix.
- Alter the matrix.

My First OpenGL Program (version 2)

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

void display (void)
{
  glClear (GL_COLOR_BUFFER_BIT);

  glBegin (GL_POLYGON);
    glVertex2f (-0.5, -0.5);
    glVertex2f (-0.5, 0.5);
    glVertex2f (0.5, 0.5);
    glVertex2f (0.5, -0.5);
  glEnd ();

  glFlush ();

  return;
}
```

My First OpenGL Program (version 2)

```
void init (void)
{
  // set clear color to black

  glClearColor (0.0, 0.0, 0.0, 0.0);

  // set fill color to white

  glColor3f (1.0, 1.0, 1.0);

  // Set up a standard orthogonal view with clipping box
  // as a cube of side length 2 centred at the origin.
  // As this is the default, these statements can be removed.

  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();

  // define the left, right, bottom and top clipping planes
  gluOrtho2D (-2.0, 2.0, -1.0, 1.0);

  return;
}
```

My First OpenGL Program (version 2)

```
int main (int argc, char **argv)
{

  glutInit (&argc, argv);

  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);

  // window size relates directly to (is proportional to)
  // the clipping planes
  glutInitWindowSize (500, 250);
  glutInitWindowPosition (0, 0);

  glutCreateWindow ("simple2");
  glutDisplayFunc (display);
  init ();
  glutMainLoop ();

}
```

Primitives

Primitives in OpenGL are the fundamental entities from which we build scenes. Primitives can be geometric and non-geometric. There are 3 geometric primitives in OpenGL:

- points
- line segments
- polygons

Primitives

Primitives in OpenGL are the fundamental entities from which we build scenes. Primitives can be geometric and non-geometric. There are 3 geometric primitives in OpenGL:

- points
- line segments
- polygons

Each geometric primitive is defined by one or more vertices. The `glBegin` function specifies the type of primitive which is then followed by a list of vertices that define the primitive. The specification of a geometric primitives ends with the `glEnd` function.

Primitives

Primitives in OpenGL are the fundamental entities from which we build scenes. Primitives can be geometric and non-geometric. There are 3 geometric primitives in OpenGL:

- points
- line segments
- polygons

Each geometric primitive is defined by one or more vertices. The `glBegin` function specifies the type of primitive which is then followed by a list of vertices that define the primitive. The specification of a geometric primitives ends with the `glEnd` function.

There are non-geometric primitives such as bitmaps.

Attributes

Attributes affect the way a primitive is rendered. While it is natural to think of the attributes as belonging to the primitive, they are actually part of the OpenGL state. When an attribute is changed it affects the way every subsequent primitive is rendered.

Points

Points are the most basic of primitives. The attributes that affect points are the point size and the colour.

```
glPointSize (2.0);
glBegin (GL_POINTS);
  glColor3f (1.0, 1.0, 1.0);
  glVertex2f (-0.5, -0.5);
  glColor3f (1.0, 1.0, 0.0);
  glVertex2f (-0.5, 0.5);
  glColor3f (0.0, 0.0, 1.0);
  glVertex2f (0.5, 0.5);
  glColor3f (0.0, 1.0, 0.0);
  glVertex2f (0.5, -0.5);
glEnd ();
```

The `glPointSize` function cannot be placed between `glBegin` and `glEnd` functions.

Lines

There are 3 types of lines.

Lines

There are 3 types of lines.

```
glBegin (GL_LINES);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.5, -0.5);
glEnd ();
```

`GL_LINES` defines a line between each successive pair of vertices. Hence the code defines a line from (-0.5,-0.5) to (-0.5, 0.5) and a line from (0.5,0.5) to (0.5,-0.5).

Lines

There are 3 types of lines.

```
glBegin (GL_LINES);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.5, -0.5);
glEnd ();
```

`GL_LINES` defines a line between each successive pair of vertices. Hence the code defines a line from (-0.5,-0.5) to (-0.5, 0.5) and a line from (0.5,0.5) to (0.5,-0.5).

```
glBegin (GL_LINE_STRIP);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.5, -0.5);
glEnd ();
```

`GL_LINE_STRIP` defines a sequence of lines that starts at the first vertex and where the end point of one segment is the start point of the next segment.

Lines (2)

```
glBegin (GL_LINE_LOOP);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.5, -0.5);
glEnd ();
```

`GL_LINE_LOOP` defines a sequence of lines that is similar to `GL_LINE_STRIP` with the addition of a line that starts at the last vertex and ends at the first vertex.

Lines (2)

```
glBegin (GL_LINE_LOOP);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.5, -0.5);
glEnd ();
```

`GL_LINE_LOOP` defines a sequence of lines that is similar to `GL_LINE_STRIP` with the addition of a line that starts at the last vertex and ends at the first vertex.

There are 3 attributes that relates to lines. `glColor*` sets the colour of any line. subsequently rendered. We will use `glName*` to denote an variant of an OpenGL function `glName`. Hence, `glColor*` denotes any variant such as `glColor3f`, `glColor4f`, `glColor3fv` etc.

Lines (3)

`void glLineWidth (GLfloat width)` sets the width, in terms of pixels, of subsequent lines.

Lines (3)

`void glLineWidth (GLfloat width)` sets the width, in terms of pixels, of subsequent lines.

`void glLineStipple (Glint factor, GLushort pattern)` sets the **stipple pattern** used to draw the line. `pattern` is a 16-bit pattern where a 0-bit means the pixel is not drawn and a 1-bit means the pixel is drawn. Successive groups of 1-bits (or 0-bits) are repeated `factor` times (up to 255). The `pattern` starts with the least significant bits.

Lines (3)

`void glLineWidth (GLfloat width)` sets the width, in terms of pixels, of subsequent lines.

`void glLineStipple (Glint factor, GLushort pattern)` sets the **stipple pattern** used to draw the line. `pattern` is a 16-bit pattern where a 0-bit means the pixel is not drawn and a 1-bit means the pixel is drawn. Successive groups of 1-bits (or 0-bits) are repeated `factor` times (up to 255). The `pattern` starts with the least significant bits.

```
glColor3f (1.0, 1.0, 0.0);
glLineWidth (2.0);
glLineStipple (3,0xcccc);
```

sets the colour of subsequent lines to yellow, their width to 2 pixels and the stipple pattern will have 6 pixels off followed by 6 pixels on.

Lines (3)

`void glLineWidth (GLfloat width)` sets the width, in terms of pixels, of subsequent lines.

`void glLineStipple (Glint factor, GLushort pattern)` sets the **stipple pattern** used to draw the line. `pattern` is a 16-bit pattern where a 0-bit means the pixel is not drawn and a 1-bit means the pixel is drawn. Successive groups of 1-bits (or 0-bits) are repeated `factor` times (up to 255). The `pattern` starts with the least significant bits.

```
glColor3f (1.0, 1.0, 0.0);
glLineWidth (2.0);
glLineStipple (3,0xcccc);
```

sets the colour of subsequent lines to yellow, their width to 2 pixels and the stipple pattern will have 6 pixels off followed by 6 pixels on.

Stippling is an OpenGL feature that needs to be specifically enabled by invoking `glEnable (GL_LINE_STIPPLE)`. It can be disabled by invoking `glDisable (GL_LINE_STIPPLE)`.

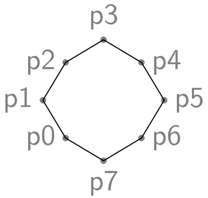
Polygons

There are 6 filled polygon primitives.

Polygons

There are 6 filled polygon primitives.

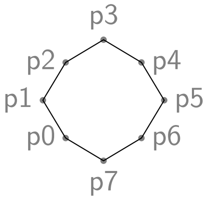
```
glBegin (GL_POLYGON);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.8, 0.0);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.0, 0.8);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.8, 0.0);
  glVertex2f (0.5, -0.5);
  glVertex2f (0.0, -0.8);
glEnd ();
```



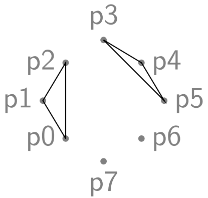
Polygons

There are 6 filled polygon primitives.

```
glBegin (GL_POLYGON);
  glVertex2f (−0.5, −0.5);
  glVertex2f (−0.8, 0.0);
  glVertex2f (−0.5, 0.5);
  glVertex2f (0.0, 0.8);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.8, 0.0);
  glVertex2f (0.5, −0.5);
  glVertex2f (0.0, −0.8);
glEnd ();
```

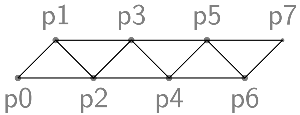



```
glBegin (GL_TRIANGES);
  glVertex2f (−0.5, −0.5);
  glVertex2f (−0.8, 0.0);
  glVertex2f (−0.5, 0.5);
  glVertex2f (0.0, 0.8);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.8, 0.0);
  glVertex2f (0.5, −0.5);
  glVertex2f (0.0, −0.8);
glEnd ();
```



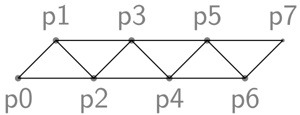
Polygons (2)

```
glBegin (GL_TRIANGE_STRIP);
  glVertex2f (0, 0);
  glVertex2f (1, 1);
  glVertex2f (2, 0);
  glVertex2f (3, 1);
  glVertex2f (4, 0);
  glVertex2f (5, 1);
  glVertex2f (6, 0);
  glVertex2f (7, 1);
glEnd ();
```

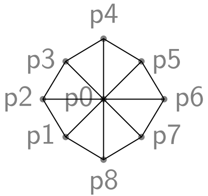


Polygons (2)

```
glBegin (GL_TRIANGE_STRIP);
  glVertex2f (0, 0);
  glVertex2f (1, 1);
  glVertex2f (2, 0);
  glVertex2f (3, 1);
  glVertex2f (4, 0);
  glVertex2f (5, 1);
  glVertex2f (6, 0);
  glVertex2f (7, 1);
glEnd ();
```

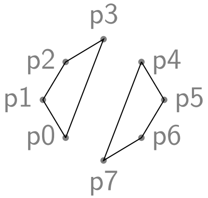



```
glBegin (GL_TRIANGE_FAN);
  glVertex2f (0.0, 0.0);
  glVertex2f (−0.5, −0.5);
  glVertex2f (−0.8, 0.0);
  glVertex2f (−0.5, 0.5);
  glVertex2f (0.0, 0.8);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.8, 0.0);
  glVertex2f (0.5, −0.5);
  glVertex2f (0.0, −0.8);
glEnd ();
```



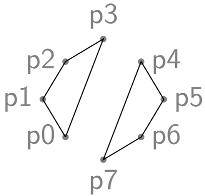
Polygons (3)

```
glBegin (GL_QUADS);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.8, 0.0);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.0, 0.8);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.8, 0.0);
  glVertex2f (0.5, -0.5);
  glVertex2f (0.0, -0.8);
glEnd ();
```

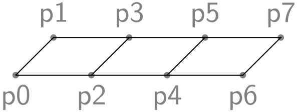


Polygons (3)

```
glBegin (GL_QUADS);
  glVertex2f (-0.5, -0.5);
  glVertex2f (-0.8, 0.0);
  glVertex2f (-0.5, 0.5);
  glVertex2f (0.0, 0.8);
  glVertex2f (0.5, 0.5);
  glVertex2f (0.8, 0.0);
  glVertex2f (0.5, -0.5);
  glVertex2f (0.0, -0.8);
glEnd ();
```




```
glBegin (GL_QUAD_STRIP);
  glVertex2f (0, 0);
  glVertex2f (1, 1);
  glVertex2f (2, 0);
  glVertex2f (3, 1);
  glVertex2f (4, 0);
  glVertex2f (5, 1);
  glVertex2f (6, 0);
  glVertex2f (7, 1);
glEnd ();
```



glRect

OpenGL provides 2 functions for drawing rectangles that are aligned with the x-y axes.

```
glRectf (type x1, type y1, type x2, type y2);
```

defines a rectange whose lower-left corner is `(x1,y1)` and upper-right corner is `(x2,y2)`.

glRect

OpenGL provides 2 functions for drawing rectangles that are aligned with the *x*-*y* axes.

```
glRectf (type x1, type y1, type x2, type y2);
```

defines a rectange whose lower-left corner is `(x1,y1)` and upper-right corner is `(x2,y2)`.

```
glRectfv (type *v1, type *v2);
```

defines a rectange whose lower-left corner and upper-right corner are pointed to by `v1` and `v2`.

Polygon Attributes

The 2 major attributes that relate to polygons are **stippling** and **colour**.

Polygon Attributes

The 2 major attributes that relate to polygons are **stippling** and **colour**.

The stipple pattern is set by:

```
void glPolygonStipple (const Glubyte *mask)
```

that sets the stipple pattern to `mask` which is a 32x32 patterns of bits. The pattern is aligned to the window and the stippling is not changed if the polygon is rotated.

Polygon Attributes

The 2 major attributes that relate to polygons are **stippling** and **colour**.

The stipple pattern is set by:

```
void glPolygonStipple (const Glubyte *mask)
```

that sets the stipple pattern to `mask` which is a 32x32 patterns of bits. The pattern is aligned to the window and the stippling is not changed if the polygon is rotated.

Colour is specified at vertices and, if the default **shading model**, `GL_SMOOTH`, is used, the interior point will be linearly interpolate between the colour of the vertices. If the `GL_FLAT` model is used, the colour of the last vertex is used.

```
glShadeModel (GLenum mode)
```

where `mode` is either `GL_SMOOTH` or `GL_FLAT`.

Saving the State

Two of the most important elements of the OpenGL state are the **model view matrix** and the **projection matrix**. These matrices can be saved and restored from **matrix stacks** using the `glPushMatrix ()` and `glPopMatrix ()` functions. Which **matrix stack** is used is determined by the OpenGL function `glMatrixMode (mode)` where `mode` can be either `GL_MODELVIEW` or `GL_PROJECTION`.

```
glMatrixMode (GL_PROJECTION);
// set projection matrix and draw scene
glPushMatrix ();
// change projection matrix and draw scene.
glPopMatrix ();
// restore previous projection matrix.
```

Saving the State (2)

OpenGL attributes can also be saved and restore from a separate attribute stack. OpenGL divides the attributes into 20 groups. For example, all the line attributes are in the group `GL_LINE_BIT` and all the polygon attributes are in the group `GL_POLYGON_BIT`. To save current attributes onto the attribute stack use:

```
void glPushAttrib (GLbitfield mask)
```

where `mask` is a logical or-ing of the attribute groups that you wish to save on the attribute stack.

Saving the State (2)

OpenGL attributes can also be saved and restore from a separate attribute stack. OpenGL divides the attributes into 20 groups. For example, all the line attributes are in the group `GL_LINE_BIT` and all the polygon attributes are in the group `GL_POLYGON_BIT`. To save current attributes onto the attribute stack use:

```
void glPushAttrib (GLbitfield mask)
```

where `mask` is a logical or-ing of the attribute groups that you wish to save on the attribute stack.

Attributes are restored by invoking:

```
void glPopAttrib ();
```