

COMP3201 – Computer Graphics

Module 1: Introduction and Graphics Primitives

1.3 Drawing Primitives

1.3.1 Drawing in OpenGL

All types of drawing in OpenGL is based on geometric primitives, which include points, lines and polygons.

A point in space is defined by a vertex; in 2D-space, we can just specify (x,y) , while (x,y,z) defines a point in 3D.

OpenGL uses 4 coordinates to define a point (x,y,z,w) – here, x , y and z are in “world coordinates”, and w is a scaling factor and is usually 1. The use of 4 coordinates (where the fourth coordinate is a scaling factor) to define 3D space is known as using homogeneous coordinates (more on this later).

“World coordinates” refers to the coordinate system we use to define the positions of objects for our scene (more on this later).

In world coordinates, the positive x -axis points to the right, positive y is upright, and the positive z -axis comes out of the screen towards you.

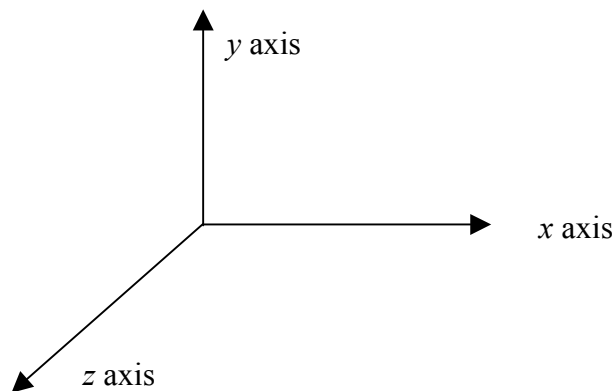


Figure 1.2

A vertex is specified using **glVertex***. This can take either scalar or vector form, and the * is replaced by a digit (2, 3 or 4) to indicate the dimension plus a character (e.g. **s**, **i**, **f**, **d**) to indicate the data type (e.g. short, integer, floating point, double respectively), plus **v** for vector, if appropriate.

For example,

glVertex2f(x, y), or **glVertex2fv(v)** in vector form;
glVertex3d(x, y, z) or **glVertex3dv(v)** in vector form.

If the z coordinate is omitted, the default value of zero is used. Also, if the fourth coordinate (the scaling factor w) is omitted, it is assumed to have the value 1.

1.3.2 Drawing Modes

There are ten modes of drawing within OpenGL.

GL_POINTS	points
GL_LINES	disconnected pairwise line segments
GL_LINE_STRIP	connected line segments
GL_LINE_LOOP	closed loop of line segments
GL_POLYGON	polygon
GL_TRIANGLES	disconnected triangles
GL_TRIANGLE_STRIP	connected triangles in a strip
GL_TRIANGLE_FAN	connected triangles in a fan
GL_QUADS	disconnected quadrilaterals
GL_QUAD_STRIP	connected quadrilaterals

1.3.3 Drawing Points

To draw a point or a series of points, list the required vertices within **glBegin(GL_POINTS)** and **glEnd()**; for example,

```
glBegin(GL_POINTS);
    glVertex2f(0.0, 0.0);
    glVertex2f(1.0, 0.0);
    glVertex2f(1.0, 1.0);
    glVertex2f(0.0, 1.0);
glEnd();
```

Note that pairs of **glBegin()** / **glEnd()** cannot be nested.

Size of Points

The size of the points can be increased by calling the function **glPointSize(3.0)** – this would set the points to be 3 pixels wide. This function call must be outside the **glBegin()** / **glEnd()** block. Point size can also be fractional (e.g. 2.4).

Note also that a point is actually a square, unless anti-aliasing is ON in which case it is represented as a circle. (Anti-aliasing will be covered later.)

1.3.4 Colour

Consider the colour cube constructed on a red/green/blue axis system. Each corner of the cube represents the colour indicated, and any other colour can be obtained by varying the intensities of the colour components.

The values here represent the range from 0% to 100% full saturation of a colour component. The OpenGL function **glColor3f** uses the range 0 to 1 to represent the colour intensities. However it is also possible to use other data types (e.g. **glColor3ub** uses unsigned bytes which range from 0 to 255). Internally OpenGL always uses the range [0, 1] to represent the colour intensities.

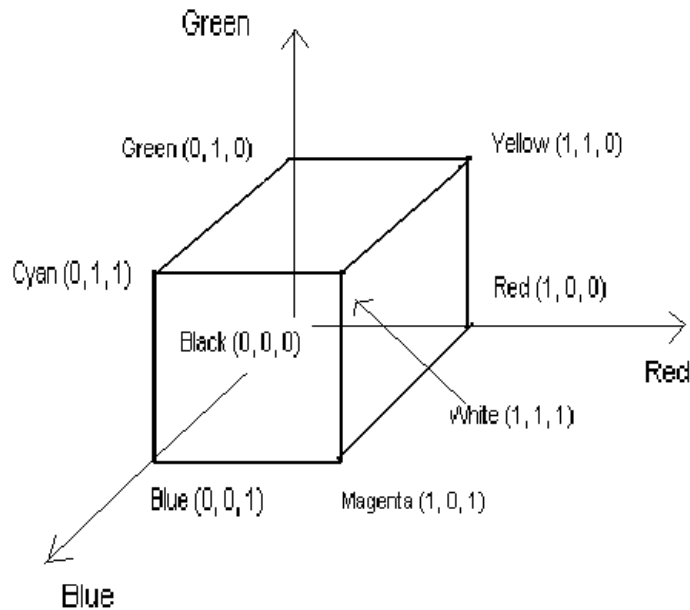


Figure 1.3

As OpenGL is a state machine, the colour specified continues to be used until it is changed by a call to **glColor*()**. Colour is represented internally by 4 values, with the default alpha-blending value of 1.0.

In this course, we will use **glColor3f()** or **glColor3fv()** with floating-point values specifying the colour. Some examples are:

glColor3f(1.0, 0.0, 0.0)	red
glColor3f(0.0, 1.0, 0.0)	green
glColor3f(0.0, 0.0, 1.0)	blue
glColor3f(1.0, 1.0, 1.0)	white
glColor3f(0.0, 0.0, 0.0)	black
glColor3f(0.0, 1.0, 1.0)	cyan
glColor3f(1.0, 1.0, 0.0)	yellow
glColor3f(0.3, 0.25, 0.8)	a shade of blue

1.3.5 Drawing Lines

The geometric primitive ‘line’ is formed by specifying two vertices. All vertices that appear between **glBegin(GL_LINES)** and **glEnd()** are read pair-wise – that is, the first two vertices define the first line, vertices 3 and 4 define the second line, vertices 5 and 6 define the third line, and so on.

The line width is specified (outside the **glBegin()/glEnd()** block) as follows:
glLineWidth(3.0).

The width parameter (which is measured in pixels) is a floating point number; the value is rounded to the nearest integer if anti-aliasing is not being used.

1.3.6 Drawing a Line Strip and Line Loop

Line segments are defined by **GL_LINES**. To create a connected series of line segments, use **GL_LINE_STRIP** – once the first segment has been specified by two vertices, any subsequent segment is defined by specifying an additional vertex. For example

```
glBegin(GL_LINE_STRIP);  
  glVertex2f(-1.8, -1.0);  
  glVertex2f(-0.4, -1.0);  
  glVertex2f(-0.4, 1.0);  
glEnd();
```

produces the left-hand object of Figure 1.4.

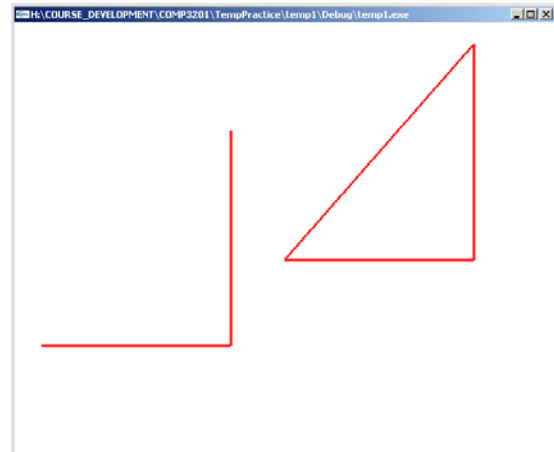


Figure 1.4

The line loop is drawn using **GL_LINE_LOOP**, where the first vertex is joined to the last vertex to form a closed loop. The following code produces the triangle in Figure 1.4.

```
glBegin(GL_LINE_LOOP);  
  glVertex2f(0.0, -0.2);  
  glVertex2f(1.4, -0.2);  
  glVertex2f(1.4, 1.8);  
glEnd();
```

1.3.7 Drawing Polygons

A polygon is a closed loop that satisfies certain constraints. A polygon can be filled, while a line loop can't, but both techniques can be used to describe the edges of objects.

```
glBegin(GL_POLYGON);  
  glVertex2f(-1.8, 0.0);  
  glVertex2f(-1.0, -1.0);  
  glVertex2f(1.0, -1.0);  
  glVertex2f(1.0, 1.0);  
  glVertex2f(0.4, 1.8);  
glEnd();
```

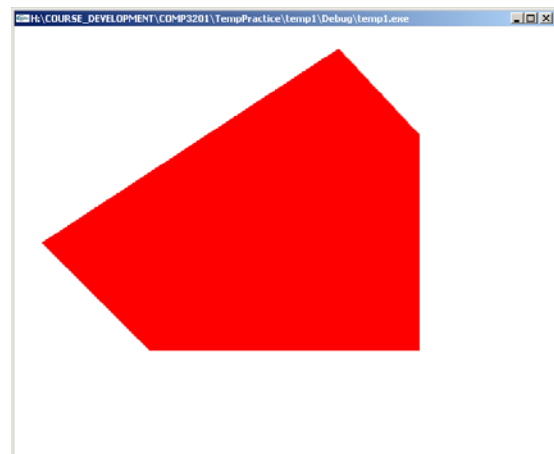


Figure 1.5

Note that a polygon has two faces, and that the edge around the polygon has no width. If the outline of the polygon needs to be drawn, then it must be drawn as an additional object.

Front/Back Facing

OpenGL distinguishes between “front” and “back” faces of objects. Specifying the vertices so that the polygon is drawn in a counter-clockwise direction means that the polygon is front-facing. It is possible to reverse this (so that this polygon is back-facing) by calling **glFrontFace(GL_CW)**. The default is **glFrontFace(GL_CCW)**.

It is important to be able to identify the front and back faces of a polygon, as in some situations you may wish to specify different physical properties for the back face (for example, its colour or material properties). Also, performance can be improved by not drawing the backward-facing polygons (see Performance Notes later).

For consistency, all polygons in a scene should be drawn in the same orientation, so that further operations, such as lighting, behave as expected.

Polygon Mode

Polygons do not need to be drawn as solid, filled shapes. Using the function **glPolygonMode** allows the polygon to be drawn filled, as an outline, or just as points at the vertices. It is also possible to specify different drawing modes for the front and back faces. Usage is

```
void glPolygonMode( GLenum face, GLenum mode);
```

The parameter **face** can be one of **GL_BACK**, **GL_FRONT**, or **GL_FRONT_AND_BACK**, while **mode** can be **GL_FILL**, **GL_LINE**, or **GL_POINT**. The default is that both front and back faces are drawn filled.

Polygon Restrictions

Note also that there are some restrictions to the construction of polygons.

- a polygon must be simple (which means that its edges must not intersect)
- a polygon must be coplanar (i.e. all its vertices must lie in a single plane, so it must not be twisted)
- a polygon must be convex (which means that it cannot have indentations; shapes with indentations should be constructed as a combination of polygons). A precise definition of convex states that a line segment joining any two interior points also lies in the interior of the polygon. Consequently, a polygon cannot have a hole in it.

Indeed, polygons that are not coplanar may even be non-convex when they are seen from a different viewpoint.

These restrictions are in place so that OpenGL can make use of very fast algorithms for rendering these polygons. The restrictions also ensure that the polygon is displayed correctly.

1.3.8 Drawing Rectangles

There is a special function for drawing rectangles, as this shape is often used in the modelling process. The function only needs two vertices to be specified – the lower left ($x1, y1$) and upper right ($x2, y2$) corners.

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

Note that **glRect*** must NOT be called between **glBegin()** and **glEnd()**.

1.3.9 Drawing Triangles

It is better (more efficient) to use mode **GL_TRIANGLES** rather than **GL_POLYGON**, if triangles are to be drawn. If more than one triangle is to be drawn, they should all be defined within the one **glBegin()/glEnd()** block. The vertices are taken 3 at a time to form each triangle.

```
glBegin(GL_TRIANGLES);  
glVertex2f(-1.5, -1.0);  
glVertex2f(0.5, -1.0);  
glVertex2f(-1.4, 1.0);  
glVertex2f(0.5, -0.2);  
glVertex2f(1.8, -0.3);  
glVertex2f(1.0, 1.5);  
glEnd();
```

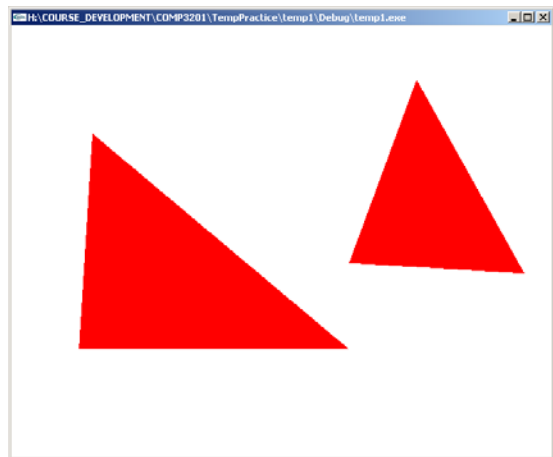


Figure 1.6

1.3.10 Drawing Quadrilaterals

Quadrilaterals are another special class of polygons, having four vertices. Multiple quadrilaterals can be drawn by including further sets of four vertices within the **glBegin() / glEnd()** block.

```
glBegin(GL_QUADS);  
glVertex2f(-1.0, -1.8);  
glVertex2f(1.6, -1.0);  
glVertex2f(0.5, 1.5);  
glVertex2f(-0.2, 1.3);  
glEnd();
```

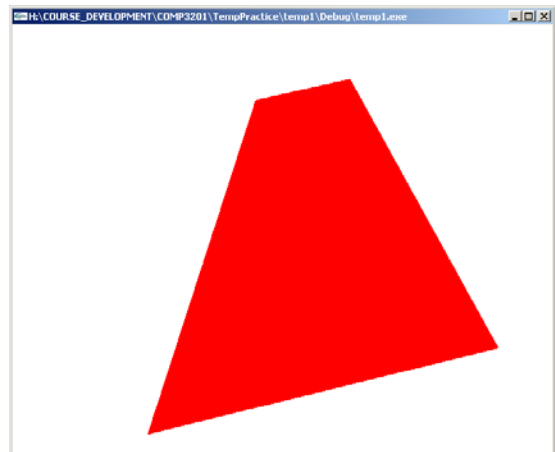


Figure 1.7

1.3.11 Strips and Fans

Often strips of triangles or quadrilaterals need to be drawn when creating a model, and an efficient way to do this is with the special GL types of **GL_TRIANGLE_STRIP**, **GL_QUAD_STRIP** and **GL_TRIANGLE_FAN**.

Figure 1.8 demonstrates a triangle strip, a triangle fan, and a quadrilateral strip. Note the order in which the vertices are used for constructing each of these objects.

For a strip of triangles, after the first triangle is specified, only one extra vertex for each extra triangle is required. The triangles are constructed in counter-clockwise order, to maintain the correct facing of the triangles.

So, for example, if the strip is specified by vertices v_0, v_1, v_2, v_3, v_4 , then the triangles are constructed as $\{v_0, v_1, v_2\}$, $\{v_2, v_1, v_3\}$, $\{v_2, v_3, v_4\}$, $\{v_4, v_3, v_5\}$, so that all the triangles are traversed in counter-clockwise direction.

Note that graphics hardware is often optimised to draw triangles fastest, and so using triangle strips is a very efficient way to construct 3D surfaces.

For a triangle fan, the first vertex specified is assumed to be the centre of the fan; each subsequent triangle uses the first and last vertices already specified together with the new vertex connection: $\{v_0, v_1, v_2\}$, $\{v_0, v_2, v_3\}$, $\{v_0, v_3, v_4\}$, $\{v_0, v_4, v_5\}$.

A strip of quadrilaterals is analogous to the triangle strip; two new vertices are required for each new quadrilateral after the first quadrilateral has been defined. In the figure below, the quadrilaterals would be constructed $\{v_0, v_1, v_3, v_2\}$, $\{v_2, v_3, v_5, v_4\}$, $\{v_4, v_5, v_7, v_6\}$, so that the quadrilaterals are traversed in a counter-clockwise direction.

Quadrilaterals must be coplanar.

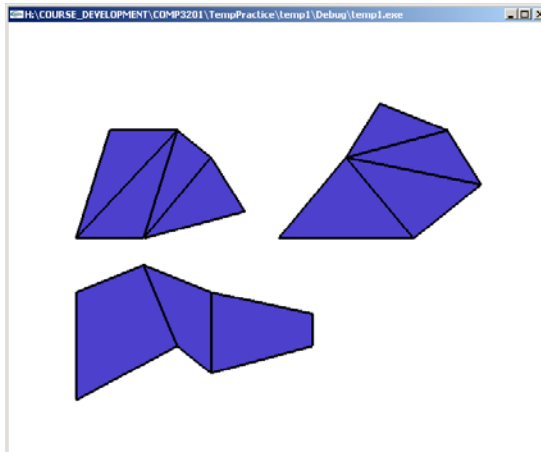


Figure 1.8

1.3.12 Colour and Shading

Colour formation in computer graphics is based on the three-colour theory, where the primary colours red, green and blue are mixed in certain quantities to form the specific colour required. The quantities range from 0.0 to 1.0, and the three primaries are always specified in Red, Green, Blue order, to correspond to what is called the

RGB Colour Model. Although this doesn't completely cover the visible electromagnetic spectrum, it is a close enough approximation.

Colours in OpenGL are set using a 4-dimensional vector, where the 4th component (which also ranges from 0.0 to 1.0) represents a transparency or opacity value. The opacity default value is 1.0, which results in a solid colour.

Here we will introduce the concepts of flat shading and interpolative shading, to allow more variety in the representation of our models.

Flat shading is fastest, and is specified by **glShadeModel(GL_FLAT)**. The colour used depends on the colour associated with the first vertex of **GL_POLYGON** or the last vertex of any other primitive (e.g. **GL_TRIANGLE_STRIP**, or **GL_TRIANGLES**). So, for example, in a triangle-strip, vertex $i+2$ (for each triangle i within the strip) determines the colour of that triangle as the strip is constructed.

Interpolative Shading (Gouraud shading) (**glShadeModel(GL_SMOOTH)**) is the default setting, and the colours are determined by interpolating between the vertices. For example, in a quadrilateral with two vertices red and two white, the colouring of the quadrilateral will range from red through pink to white at its far end.

An alternative shading model is Phong shading, but although the colour blendings are much smoother, Phong shading is computationally intensive and so is not often used in a real-time environment and is not directly supported by OpenGL.

1.3.13 Valid Commands between glBegin() and glEnd()

Only certain commands are allowed between **glBegin()** and **glEnd()**. The following table (from the OpenGL Red Book) lists these valid commands.

Valid Commands	Description
glVertex*()	specify vertex coordinates
glColor*()	set the current colour
glIndex*()	set current colour index
glNormal*()	set normal vector coordinates
glTexCoord*()	set texture coordinates
glMultiTexCoord*ARB()	set texture coordinates for multitexturing
glEdgeFlag*()	control drawing of edges

Table 1.2