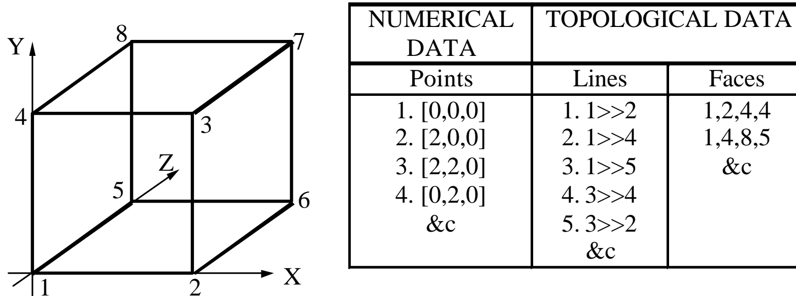
Lecture 6: Polygon rendering and OpenGL

3-Dimensional Objects Bounded by Planar Surfaces (Facets)



NUMERICAL DATA	TOPOLOGICAL DATA	
Points	Lines	Faces
1. [0,0,0] 2. [2,0,0] 3. [2,2,0] 4. [0,2,0] &c	1. 1>>2 2. 1>>4 3. 1>>5 4. 3>>4 5. 3>>2 &c	1,2,4,4 1,4,8,5 &c

Diagram 1: Representing 3D objects

A planar facet is defined by an ordered set of 3D vertices, lying on one plane, which form a closed polygon, (straight lines are drawn from each vertex to the following one with the last vertex connected to the first). The data describing a facet are of two types. First, there is the numerical data which is a list of 3D points, (3*N numbers for N points), and secondly, there is the topological data which describes which points are connected to form edges of the facet. For the cube shown in Diagram 1 we need 24 real numbers for the numerical data, 24 integers to store the line topology, and 24 integers to store the face topology. We will also need to maintain the number of lines and faces in the figure, and the number of edges per face. All this could be done using static structures (arrays), alternatively, if we start with abstract data types that express the structure of three dimensional objects, we may define the following data types (Diagram 2):

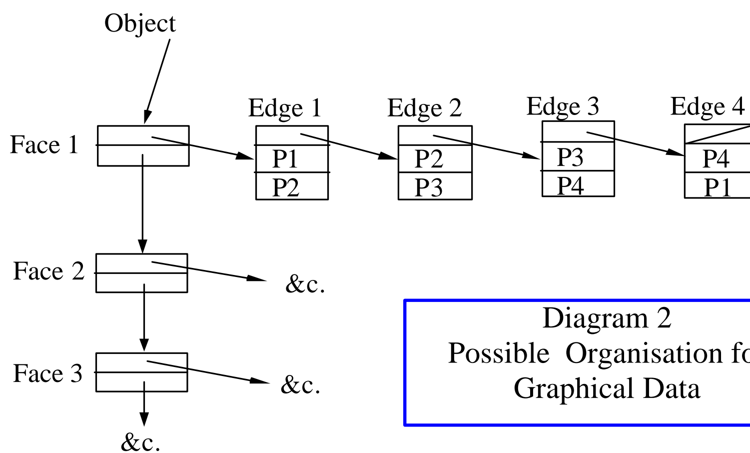


Diagram 2
Possible Organisation for
Graphical Data

```
ordinate=x,y,z;
Point = array[ordinate] of real;
Edge =      record
                start, finish: Point;
            end ;
Edgelist =  record
                thisedge: Edge ;
                nextedge: ^Edgelist;
            end ;
Facetlist = record
                thisfacet: ^Edgelist;
                nextfacet:^Facetlist;
            end ;
Object=^Facetlist;
```

It should be noted that redundancy exists in this case, since edges which belong to two facets are duplicated, and vertices which belong to three edges appear three times. However, when a large number of objects are processed, redundancy of data may help the speed. Later on in this lecture we will see other examples where this is true.

What is OpenGL?

OpenGL is a software library which creates an interface to graphics hardware. The OpenGL library provides an interface to the graphics hardware at the lowest level. Many graphics applications or graphics packages provide an interface to the graphics hardware at a much higher level and are built on top of OpenGL. Examples are packages such as OpenInventor or VTK. OpenGL is not a

general purpose graphics system. Instead, OpenGL is a polygon-based rendering system which supports a small number of geometric primitives for creating models: points, lines and polygons.

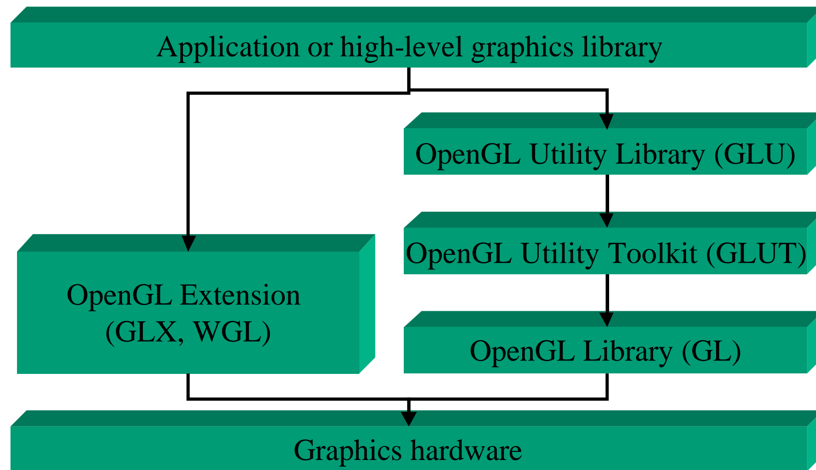


Figure 1: Overview of the OpenGL API

OpenGL is hardware and operating system independent. It is up to the underlying implementation of the OpenGL system to decide how to implement the OpenGL functionality. On some platforms, the implementation may use accelerated hardware for rendering while on other platforms the implementation may use only software for rendering. OpenGL is available for operating systems like Windows 98/NT, Linux and Unix and for a wide variety of programming languages such as C, C++, Ada, Fortran and Java.

The functionality of the OpenGL library is extended by a number of related libraries: The OpenGL utility library (GLU) contains several convenience routines which are implemented in terms of lower-level OpenGL commands. These include routines for setting up viewing transformations. The OpenGL Utility Toolkit (GLUT), a window system independent toolkit for writing OpenGL programs, implements a simple windowing application programming interface for OpenGL. GLUT makes it considerably easier to learn about and to explore OpenGL programming. In addition it provides a portable application programming interface (API) which facilitates writing a single OpenGL program that works on window systems such as Microsoft Windows and X Window System environments. Finally, each window system has a specific library that extends OpenGL functionality on that platform. For the X Window System, the OpenGL extension is called GLX. For Microsoft Windows, the OpenGL extension is called WGL.

OpenGL Command Syntax

All OpenGL commands use the prefix gl before the command name. Similarly, the OpenGL related libraries GLU and GLUT use the prefix glu and glut before the command name. For example, to define a vertex at position (1.0, 2.0, 0) the OpenGL command is:

glVertex3f(1.0, 2.0, 0.0);

In addition, the suffix 3f after the command name specifies the number of arguments and their type. In the example above, the last letter indicates the data type of the arguments (float in this case), while the preceding number indicates the number of arguments expected (3 in this case). Other

possible data types include bytes (suffix b), short integers (suffix s), long integers (suffix i), and doubles (suffix d). Most OpenGL arguments are available for different number and types of arguments. For example, the effect of the command

glVertex2i(1, 2);

is equivalent to the command above, except that it expects only two arguments whose type is int.

OpenGL State Machine

OpenGL is a state machine. OpenGL can be put into various states (or modes) that remain in effect until they are changed. The state encapsulates control for operations like transformations, rendering, lighting, shading and texture mapping. For example color is a state variable. By setting the color to red with command

glColor3f(1.0, 0.0, 0.0);

the state of the variable color is set to red and all objects are drawn in this color until the current color is set to another color. Many other state variables can be enabled with the commands **glEnable()** and **glDisable().**

OpenGL Rendering Pipeline

In OpenGL the different processing stages define the OpenGL rendering pipeline (shown in Figure 2). At the beginning of this pipeline are either vertex data (describing points, lines, and polygons) or pixel data (bitmaps, pixmaps). The rendering pipeline is slightly different for vertex and pixel data. In the following, the rendering pipeline for vertex data is described in more detail.

All geometric data has to be represented by vertices. However, it is common in computer graphics to model curves and surfaces by representing them in a parametric form, i.e. using control points. These representations are compact and easy to modify and store. Before such curves and surfaces can be rendered, they have to be transformed to vertices. Evaluators provide ways of deriving vertex data including their spatial coordinates, texture coordinates and surface normals from these parametric representations.

The vertex data is then passed to the next stage during which per-vertex operations are carried out. In this stage vertices are transformed from their world coordinates to their screen coordinates. If texture-mapping is enabled, the texture coordinates of vertices are calculated. If lighting is enabled, the lighting for each vertex is calculated using its transformed position and surface normal, the light source position, as well as other lighting information. The next stage for vertex data is the primitive assembly during which vertex data may be clipped according to the shape of the viewing volume. For point data, clipping may be achieved by accepting or rejecting vertices. For line and polygon data, clipping may introduce new vertices depending on how a line or polygon is clipped.

During the rasterization stage, both geometric data and pixel data are converted to fragments where each fragment corresponds to a pixel in the framebuffer. In this step, parameters such as line pattern and width are taken into account. Furthermore, calculations necessary for antialiasing are carried out. At the end of this step each fragment has an associated value for its color and depth.

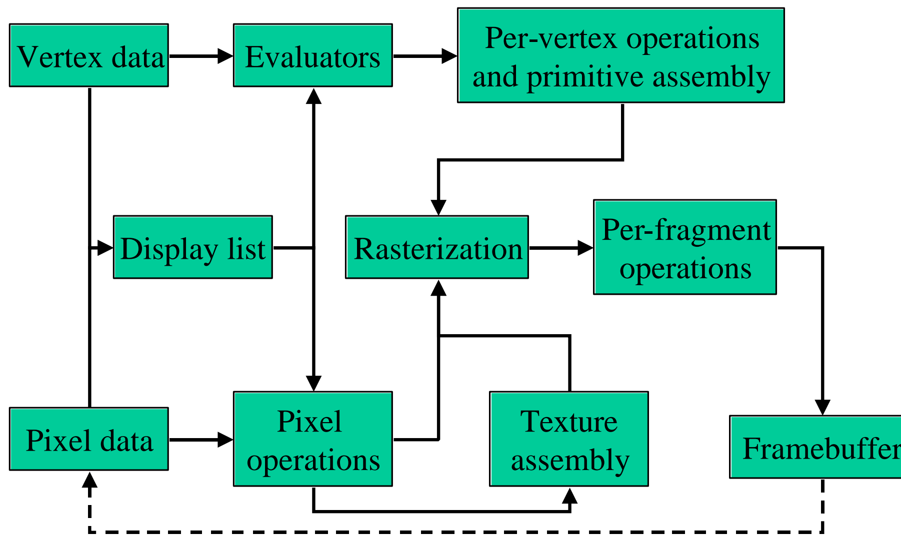


Figure 2: Order of operations in the OpenGL rendering pipeline

In the final stage, so-called per-fragment operations are performed. Before anything is drawn into the framebuffer, series of operations are performed to modify or delete fragments. For example, fragments may be passed through a number of tests including the depth-buffer test (which eliminates hidden surfaces using a z-buffer algorithm). If a fragment fails the test, it is not processed any further. Finally, other operations such as blending or dithering may be performed. At the end of this step, the processed fragments are drawn into the framebuffer.

OpenGL Drawing Primitives

OpenGL supports a number geometric primitives for drawing. All geometric primitives are described in terms of vertices. Each vertex is represented as the three coordinates (x, y, z) of a 3D point and can be used to define the endpoints of line segments or the corners of polygons. Note that internally OpenGL uses homogeneous coordinates so that each vertex is actually represented by four coordinates (x, y, z, w). If w is different from zero, these coordinates correspond to the Euclidean point (x/w, y/w, z/w).

Since vertices can be used to define different geometric primitives such as points, lines and polygons, the type of geometric primitive associated with the vertex data has to be defined explicitly. For example, the following two code segments can be used to draw a polygon or a set of points:

```
glBegin(GL_POLYGON);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 3.0, 0.0);
    glVertex3f(4.0, 3.0, 0.0);
    glVertex3f(5.0, 1.5, 0.0);
    glVertex3f(4.0, 0.0, 0.0);
```

```
glEnd();


glBegin(GL_POINTS);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(0.0, 3.0, 0.0);
    glVertex3f(4.0, 3.0, 0.0); glVertex3f(5.0, 1.5, 0.0);
    glVertex3f(4.0, 0.0, 0.0);
glEnd();
```

An example of the output of the two code segments is shown in Figure 3. In addition to these two geometric primitives, OpenGL defines eight further geometric primitives shown in Figure 4. These include unconnected line segments (**GL_LINES**), connected line segments (**GL_LINE_STRIP**), triangles (**GL_TRIANGLES**), triangle strips (**GL_TRIANGLE_STRIP**), and fans (**GL_TRIANGLE_FAN**) as well as quadrilaterals (**GL_QUADS**) and quadrilaterals strips (**GL_QUAD_STRIP**). Quadrilaterals are four sided polygons. In addition to defining the coordinates of each vertex (using **glVertex()**), one can also define additional data for each vertex such as its color (using **glColor**()), its normal (using **glNormal()**) and its texture coordinates (using **glTexCoord**()).

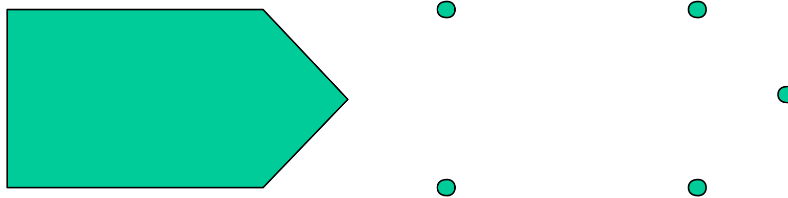


Figure 3: Drawing a polygon or a set of points

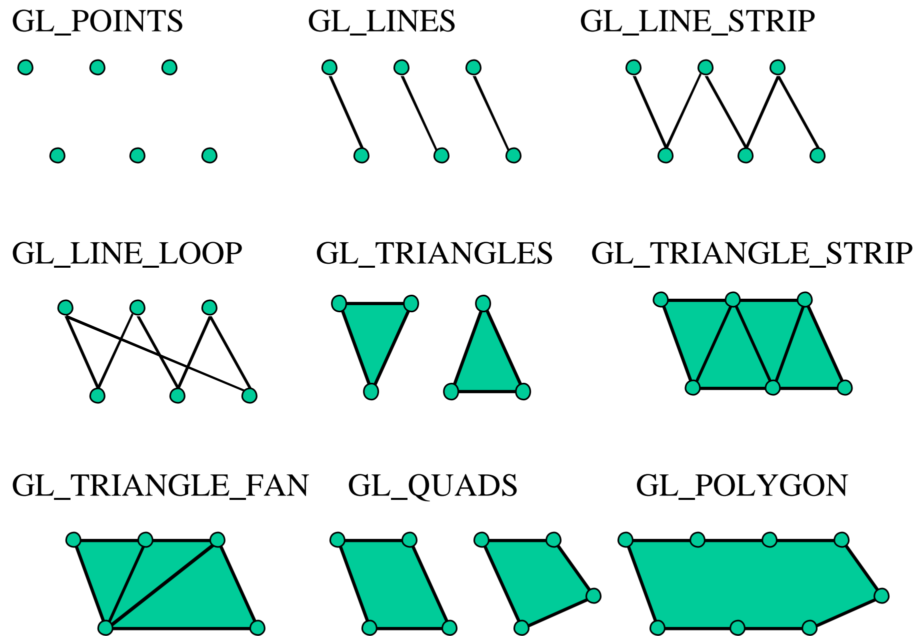


Figure 4: Different geometric primitives

In general, polygons can be very complicated and slow to draw so that many high performance graphics systems like OpenGL make some strong assumptions on *valid* polygons. First, the edges of the polygon cannot intersect (i.e. the polygon must be *simple*). Second, polygons cannot have indentations (i.e. the polygon must be *convex*). In the real-world, the surfaces of objects consist often of non-simple or non-convex polygons. However, since all non-simple or non-convex polygons can be described as unions of simple, convex polygons, the problem does not arise in practice. For example, to draw non-convex polygons, one can subdivide them into convex polygons, i.e. triangles. Finally, the polygons must be *planar*, i.e. the vertices of a polygon must lie in a plane. Non-planar polygons are not allowed since after various transformations they may project to non-simple or non-convex polygons.

OpenGL Viewing

In order to render geometric primitives, OpenGL requires the definition of a number of transformations which determine the relationship between world, model, camera and screen coordinates. In OpenGL, the viewing, modelling, and projection transformations are specified by 4 x 4 matrices. The first matrix is the *modelview* matrix which transforms object coordinates into eye coordinates. In the next step, OpenGL applies the projection matrix to yield clipped coordinates. At this point, normalized device coordinates are obtained by perspective division, that means coordinates in the form (x, y, z, w) are normalised to yield (x/w, y/w, z/w, 1). Finally, window coordinates are obtained by applying the viewport transformation.

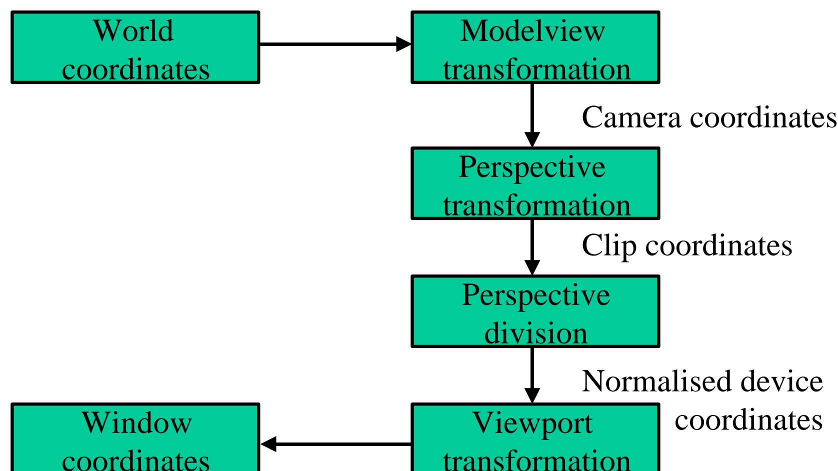


Figure 5: Transformtions in OpenGL

The easiest way to set up the viewing transformation is to use the function gluLookAt():

void gluLookAt(float eyex, float eyey, float eyez, float focalx, float focaly, float focalz, float upx, float upy, float upz);

Here the desired viewpoint is specified by eyex, eyey and eyez. The focalx, focaly and focalz arguments specify the focal point along the line of sight while the upx, upy and upz arguments specify the direction which is up.

OpenGL supports perspective and orthographic (parallel) projection transformations. In order to define a perspective transformation, a viewing frustum must be defined. This viewing frustum forms a truncated pyramid which encapsulates the viewing volume (the part of the world which is visible). In this truncated pyramid the base corresponds to the front clipping plane while the apex corresponds to the back clipping plane.

The easiest way to set up the perspective projection transformation is to use the function gluPerspective():

void gluPerspective(float fovy, float aspect, float near, float far);

The meaning of the parameters of the function gluPerspective() is illustrated in Figure 6: The parameter fovy specifies the angle of the field of view while the aspect defines the aspect ratio of the viewing frustum, i.e. its width divided by its height. The near and far values denote the distance of the viewpoint to the front and back clipping planes.

As mentioned it is also possible to set up a parallel (or orthographic) projection transformation using the function glOrtho:

glOrtho(double left, double right, double bottom, double top, double near, double far);

The meaning of the parameters of the function glOrtho() is illustrated in Figure 7.

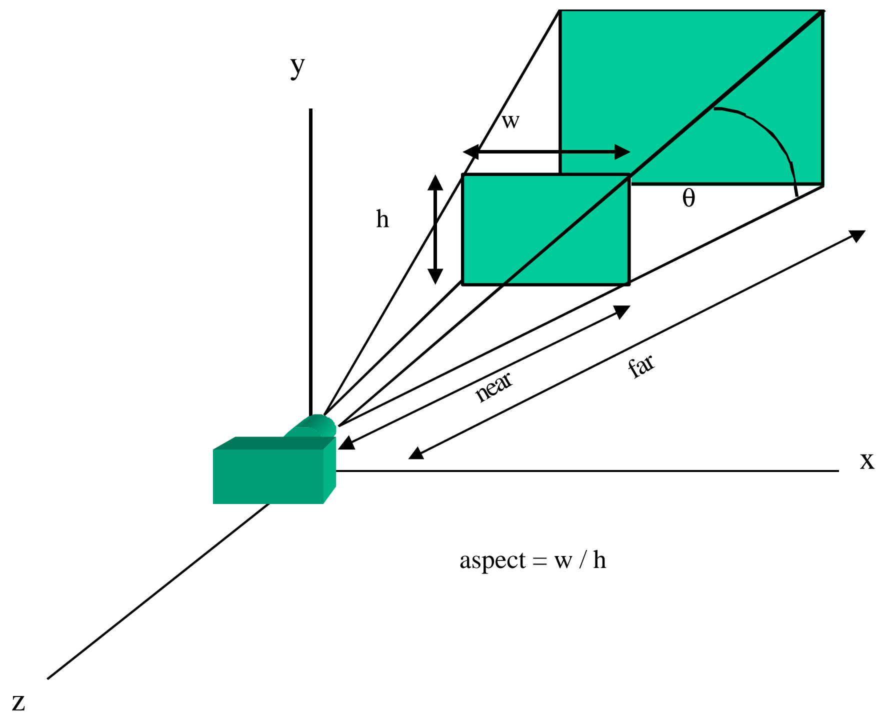


Figure 6: Specifying the perspective projection transformation using gluPerspective

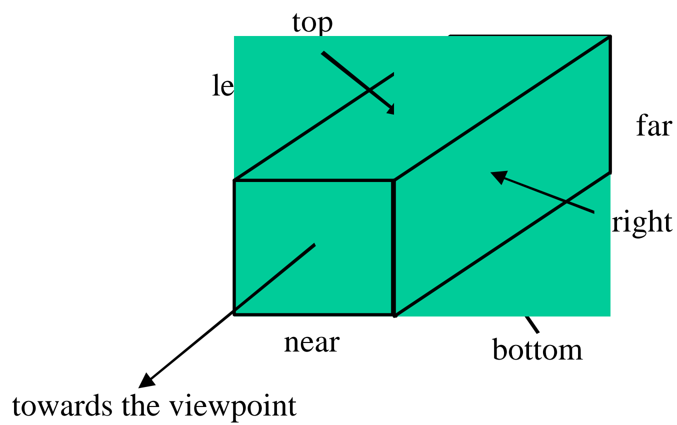


Figure 7: Specifying the parallel projection transformation using glOrtho