Monash University
CSE3313 Computer Graphics

Tutorial 5 2007

1 October 2007

Objectives

- Understand different methods of drawing geometry in OpenGL;
- Understand how to use vertex arrays to send faceted data to OpenGL;
- Gain knowledge about the *winged-edge* data structure used to represent polygonal objects;
- Simple lighting models in OpenGL.

Geometric Representation

> "OpenGL is principally concerned with object rendering; it does not provide explicit support for creating object models. The model input data is left for the application to provide."
>
> — *Advanced Graphics Programming Using OpenGL* [2]

These notes look at how to model and represent basic geometric structures using OpenGL.

Drawing Polygons

To date in CSE3313 Computer Graphics you were taught to draw polygons like this:

```
glBegin(GL_POLYGON);
 for (i = 0; i < NUM_POINTS; ++i)
 {
   glColour3f(colours[i].x, colours[i].y, colours[i].z);
   glNormal3f(normals[i].x, normals[i].y, normals[i].z);
   glVertex3f(vertices[i].x, verticies[i].y, verticies[i].z);
 }
glEnd()
```

While this is easy to understand it is not efficient. We covered additional 'strip' primitives such as `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` and `GL_QUAD_STRIP` which allow mesh strips and caps to be defined. We also saw that you could use calls such as `glVertex3fv()` to send arrays of vertices for a single primitive, eliminating the need for so many OpenGL calls necessary to define an object (eliminating the `for` loop in the above code). Function calls incur an overhead so can adversely affect performance, particularly if we are sending large datasets to the GPU.

Display lists are a simple method of managing compound graphics objects, but those objects still need to be defined. In many instances, we don't just want to render a few simple polygons, but complex objects represented by *polygonal* or *triangular meshes.* Meshes have the property that most edges (hence vertices) are shared between faces. If we were to define these shapes individually using calls to `glVertex*()` there would be a lot of redundancy.

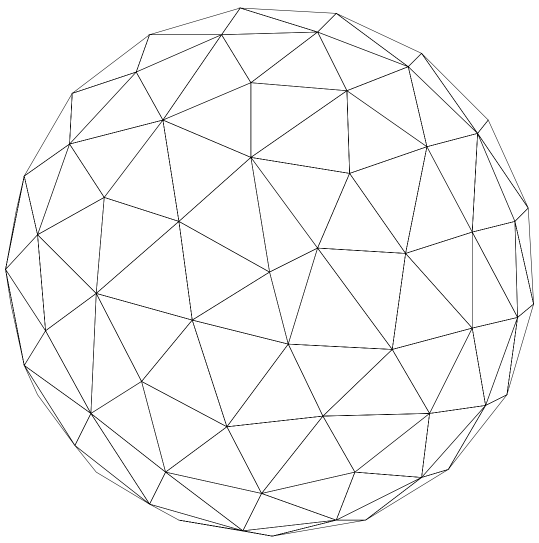

Figure 1: A triangulated representation of a sphere. All vertices are shared by multiple triangles

Faceted Model Checklist

Here is a basic check list of things you need to generate and check when creating geometric models for OpenGL.

1. For shaded and textured models, you need to generate:

vertex co-ordinates: a scalar triple (x, y, z). Most cards use a float representation internally so this is the preferred format.

vertex normals: these should be normalised and generated on a per-vertex basis. 'Hard' edges will require different normals for shared geometric vertices along the edge (hence duplicated vertices). Facet (polygon) normals can be computed by taking the cross product of two vectors formed from the two sets of vertex pairs that are furthest apart from each other. Vertex normals can be generated by averaging the facet normals for each polygon that the vertex shares.

texture co-ordinates: for 2D texture mapping, texture co-ordinates (s, t) must be associated with each geometric vertex. OpenGL uses 4 dimensional texture co-ordinates (s, t, r, q) and 3D textures have been possible since OpenGL 1.2. The default value for r and q is 0. Note that just like geometric vertices, texture co-ordinates can be transformed by OpenGL allowing you to move, rotate, scale textures over an object.

2. Ensure that the *vertex winding order* is consistent (clockwise or counter-clockwise). CCW is the standard convention. If all polygons have the same winding order, this allows you to exploit backface removal in OpenGL.

3. Use compound primitives such as triangle strips where possible. You can even do compound structures using *greedy tri-stripping* [2, page 13].

Vertex Arrays

Version 1.1 of OpenGL introduced *vertex arrays* which are an efficient mechanism for sending large numbers of vertices, colours, texture vertices and normals. Version 1.4 extended the vertex array concept to allow storing of fog co-ordinates and secondary colours in vertex arrays.

The steps to drawing a vertex array are as follows:

1. Enable up to eight arrays, each storing a different type of data (for example vertices, normals, colours).

2. Put the data into arrays. The data will be accessed via pointers passed in OpenGL calls.

3. Draw the geometry using the data. OpenGL obtains the data from the supplied vertex arrays.

Enabling Client State

The first step is to activate the appropriate arrays using the call:

```
glEnableClientState(GLenum type)
```

This function takes an enumerated parameter that enables the appropriate array. The most common arrays you're likely to use are vertex, normal and texture co-ordinates. For example:

```
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

The corresponding call to `glDisableClientState()` will disable the use of that array. Note that these calls *cannot* be stored in a display list.

Specifying Data

There are eight different calls for specifying the data depending on the type of data in the array. To specify vertices use:

```
void glVertexPointer(GLint size, GLenum type,
                     GLsizei stride, const GLvoid * data);
```

`size` is the number of co-ordinates per vertex and must be 2, 3, or 4. `type` is the data type (one of `GL_SHORT`, `GL_INT`, `GL_FLOAT`, `GL_DOUBLE`). `stride` is the byte offset between consecutive vertices. This would normally be 0 for a tightly packed array. Other values are useful when the data is interleaved (i.e. different types mixed together in one array).

Drawing

Enabling arrays and specifying data does not yet draw anything, or send any data to the graphics subsystem. The elements need to be extracted from the arrays in some way. The simplest, random access method is `glArrayElement(GLint i)` which obtains the data of the i'th vertex for all currently enabled arrays. So if you have enabled colour, normals and verticies, a single call to `glArrayElement()` specifies all three in a single call.

`glDrawElements()` is similar, but assumes the array data has some logical ordering. This allows you to reduce the drawing of an individual primitive to a single call. The function prototype is:

```
glDrawElements(Glenum mode, GLsizei count, GLenum type,
               void * indices)
```

where `mode` specifies the kind of primitive you want to use to do the drawing: this is one of the values accepted by `glBegin()` such as `GL_POLYGON` or `GL_LINE_LOOP`. `count` is the number of elements, which will be found in the array pointed to by `indices`. `type` specifies the type of data in the array: it must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`. Obviously space considerations and total number of indices will come into play here to determine the appropriate type for each index.

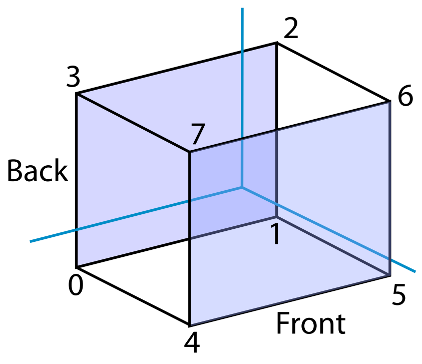

Figure 2: A cube showing vertex numbering

Here is an example — drawing a cube. We first define arrays of vertices and colours:

```
static GLfloat vertices[] = {-1.0, -1.0, -1.0,
                              1.0, -1.0, -1.0,
                              1.0,  1.0, -1.0,
                             -1.0,  1.0, -1.0,
                             -1.0, -1.0,  1.0,
                              1.0, -1.0,  1.0,
                              1.0,  1.0,  1.0,
                             -1.0,  1.0,  1.0
                            };
static GLfloat colours[] = { 0.0, 0.0, 0.0,
                             1.0, 0.0, 0.0,
                             1.0, 1.0, 0.0,
                             0.0, 1.0, 0.0,
                             0.0, 0.0, 1.0,
                             1.0, 0.0, 1.0,
                             1.0, 1.0, 1.0,
                             0.0, 1.0, 1.0
                           };
```

We could also define normals if necessary at this stage too. Next we define the connectivity for each face of the cube:

```
static GLubyte frontIdx[] = {4, 5, 6, 7};
static GLubyte rightIdx[] = {1, 2, 6, 5};
static GLubyte bottomIdx[] = {0, 1, 5, 4};
static GLubyte backIdx[] = {0, 3, 2, 1};
static GLubyte leftIdx[] = {0, 4, 7, 3};
static GLubyte topIdx[] = {2, 3, 7, 6};
```

The next step is to set up the arrays. This needs to be done once only.

```
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);

    glColorPointer(3, GL_FLOAT, 0, colours);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
```

We can now draw each face with a single call to `glDrawElements()`.

```
    /* draw the cube using individual GL_QUADS:
        each element is a polygon */
    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIdx);
    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIdx);
    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIdx);
    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, backIdx);
    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIdx);
    glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIdx);
```

As we are drawing `GL_QUADS` we can concatenate the entire connectivity list and reduce the cube definition to a single call to `glDrawElements()`.

```
static GLubyte allIdx[] = {4, 5, 6, 7,
                           1, 2, 6, 5,
                           0, 1, 5, 4,
                           0, 3, 2, 1,
                           0,4, 7, 3,
                           2, 3, 7, 6
                          };

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIdx);
```

This would not work is we'd used `GL_POLYGON` rather than `GL_QUADS`. Do you know why?

`glMultiDrawElements()` replaces a number of calls to `glDrawElements()` in a single call. The prototype is as follows.

```
glMultiDrawElements(GLenum mode, const GLsizei * count,
                    GLenum type, const GLvoid * * indices,
                    GLsizei primcount)
```

where `mode` and `type` have the same function as for `glDrawElements()`, `count` is an array of vertex counts for each respective array element list. `primCount` is the number of elements to be drawn.

Here's how we would use it for the cube.

```
static GLubyte * indices[] = { frontIdx,
                               rightIdx,
                               bottomIdx,
                               backIdx,
                               leftIdx,
                               topIdx };

static GLsizei counts[] = {4, 4, 4, 4, 4, 4 };

glMultiDrawElements(GL_QUADS, counts,
                    GL_UNSIGNED_BYTE,
                    (const GLvoid * *)indices, 6);
```

Sequenced Arrays

In the examples above, arrays of indices were required to specify which particular element of the vertex array should be used for each vertex that makes up the primitive. Additional OpenGL calls are available if you just wish to 'sequence through' your vertex data in linear order. This eliminates the need for arrays of indices as used in the previous examples.

The function

```
glDrawArrays(GLenum mode, GLint first, GLsizei count)
```

constructs a sequence of geometric primitives of type `mode` using the array elements starting at `first` and ending at `first + count - 1` of each enabled array.

`glMultiDrawArrays()`, introduced in OpenGL 1.4 parallels the function `glMultiDrawElements()`, combining several `glDrawArray()` calls into a single call. The prototype is

```
glMultiDrawArrays(GLenum mode, const GLint * first,
                  const GLsizei * count, GLsizei primcount)
```

`first` and `count` contain lists of array locations indicating where to process each list of array elements. See the *OpenGL programming guide* (the red book) for more details.

Summary

Vertex arrays, used in combination with display lists, are a fast and efficient way to send polygonal meshes (and other geometric data) to the GPU with a minimum of overheads. Modern graphics cards are well equipped to take advantage of these features. I have placed the simple cube demo used in this paper on the wiki for you to play with.

A typical example of where vertex arrays would be useful for this project is in describing terrain data. Terrain data can be specified as a *height field* (a 2D array of height values for regular samples over a plane). It should be easy to generate vertex arrays using heights found from combinations of the `noise()`, `dnoise()`, `fractalSum()` and `turbulence()` functions, for example. These functions can be used to generate geometric vertex, normal, colour, texture, and maybe even fog arrays that can be sent to OpenGL as triangles[1].

Vertex Buffer Objects

The main problem in getting better performance in modern graphics cards is in moving geometry data from main memory to the GPU. Display lists give the option of keeping the geometric data in GPU memory so it doesn't need to be constantly transferred at each redraw.

Vertex arrays manage memory on the CPU side of the CPU/GPU divide (in the client's address space). OpenGL 1.5 adds the concept of *vertex buffer objects* to enable the efficiencies of vertex arrays on the server side, just like display lists. So rather than using main memory, vertex data is obtained from the GPU side. The application can explicitly transfer or map vertex array data using the `glBufferData` and `glMapBuffer` functions respectively. Just as with normal vertex arrays, the data can be modified (to allow animation for example), however read/write access requires shifting data between main memory and GPU memory. Vertex buffer objects can be used to get maximum performance from modern graphics cards.

Data Structures for Polygonal Meshes

A large number of data structures have been developed to store and manipulate polygonal meshes. Here I quickly review a few common ones.

The simplest representation for a triangle mesh is to store a list of m triples of $3m$ vertices. This is known as a *triangle soup*. It's not particularly efficient for representing structures with shared edges, although easy from an implementation and management perspective.

We'll now look at a couple of polygonal data structures, useful for storing and manipulating compound meshes. The material is primarily taken from

[1] Using `GL_QUADS` is not advisable as these quads will not be planar.

the book by O'Rourke [3].

Winged-Edge Data Structure

One of the first data structures for polyhedral boundary representation is the *winged-edge* representation developed by Baumgart [1]. The main focus of this structure is the edge. Each object maintains three rings: em vertices, edges and *faces*. Each vertex points to an arbitrary one of its incident edges, each face to an arbitary one of its bounding edges. Each edge record, e, consists of eight pointers: the two verticies v_0 and v_1 forming the edge; two faces shared by e, f_0 and f_1 (left and right according to the direction of $v_0 \to v_1$); the two incoming edges e_0^- and e_0^+; the two outgoing edges e_1^- and e_1^+. These four edges form the 'wings' of e (see Figure 3). Note the direction of incoming edges $e_0^- \to e_0^+$ and $e_1^- \to e_1^+$ is a clockwise ordering.

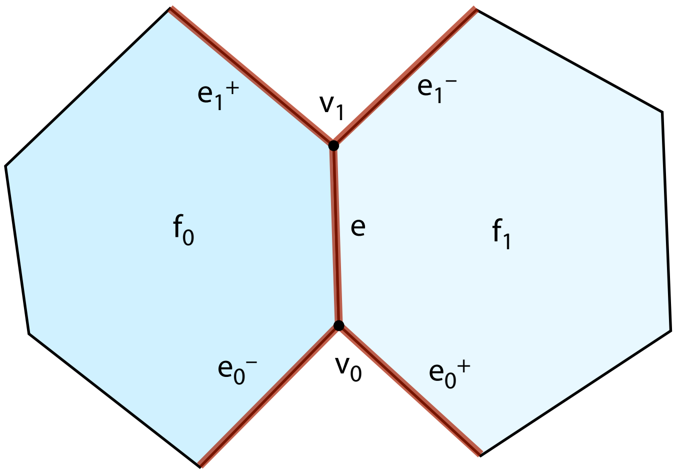

Figure 3: The winged-edge data structure. From [3, page 146].

O'Roukre gives an example use of this data structure: to find all the edges bounding face f, retrieve the sole edge record pointed to by f and follow the e^+ edges around f until e is encountered again. However, because e is ordered arbitrarily, it is necessary to check if f is left or right of e to decide whether the e_1^+ or e_0^+ edge should be followed.

The *twin-edge* data structure improves on the winged-edge method by allocating each edge as two opposing half edges. This increase in storage space makes traversal and modification operations easier and faster. The *quad-edge* data structure is slightly more complex but extremely general in terms of representation (see [3] for details). These data structures are not only used for geometric modelling, but for other entities such as graphs.

The main reason to use these data-structures is if your geometric data is being edited or changed. For something like a regular height-field it would probably be unnecessary. In many interactive graphics applications, several different representations of the same object are maintained (for example, for physics simulation or view-frustrum culling a low-resolution model may be used, while a more detailed version used for display).

References

[1] Bruce Guenther Baumgart. A polyhedron representation for computer vision. *Proceedings AFIPS National Computer Conference*, 44:589–596, 1975. 9

[2] Tom McReynolds and David Blythe. *Advanced graphics programming using openGL*. The Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier Morgan Kaufmann Publishers, San Francisco, CA, 2005. 1, 3

[3] Joseph O'Rourke. *Computational geometry in C*. Cambridge University Press, Cambridge, UK; New York, NY, USA, 2nd edition, 1998. 9