

Vertex Arrays

Notes based on

Shreiner, D., Woo, M., Neider, J. and Davis, T., (2006), OpenGL Programming Guide, 5th Edition, Addison Wesley

McReynolds, T. and Blythe, D. (2005), Advanced Graphics Programming Using OpenGL, Morgan Kaufman.

Introduction

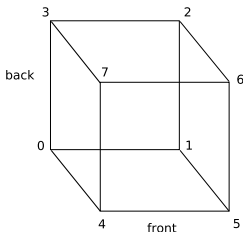
- ▶ 3D graphics geometric models can be specified in various ways.
- ▶ Polygonal modelling where objects are defined by sets of polygons, and in turn vertices, is the most common.
- ▶ OpenGL, and its rendering pipeline, primarily uses a primarily polygonal approach, although there is support for higher order surfaces.
- ▶ OpenGL has a small number of primitives for modelling 2D and 3D objects: points, lines, triangles, quadrilaterals, and general convex polygons.
- ▶ There are also OpenGL primitives specifically to improve performance for *polygon meshes*.

Polygon Meshes

- ▶ In polygon meshes, or just *meshes*, polygons share vertices and edges.
- ▶ Eliminating repetition of redundant shared vertex information is a way of improving performance.
- ▶ OpenGL has strip primitives, e.g., `GL_TRIANGLE_STRIP` and `GL_QUAD_STRIP` to improve performance for meshes, which remove some repetition.
- ▶ Strips usually require special processing of meshes, as meshes are not normally stored as strips.
- ▶ Another approach to improving performance for meshes is the use of *vertex arrays*.

Cube Example

A cube has 8 vertices, 12 edges and 6 faces:



- ▶ Each vertex is shared between 3 faces.
- ▶ Using quads, i.e., `glBegin(GL_QUADS)` and `glVertex` to specify the cube gives 6 quads with 24 vertices.
- ▶ However, really only 8 vertices!

Performance and Architectural Considerations

- ▶ Redundant vertices still require *transfer* and *processing* of the data
- ▶ Similar considerations for vertex colours and normals.
- ▶ Usually a graphics accelerator, typically a GPU, sits on a fast bus, e.g., PCI express (PCIe) and has its own memory.
- ▶ Even with a fast bus, data must be transferred from system memory to the GPU for processing.
- ▶ Transferring data can become the bottleneck, particularly when millions of polygons per frame are involved.

Vertex Arrays

- ▶ Basic idea: regard meshes as a primitive.
- ▶ Transfer and process as a group vertices belong to a mesh.
- ▶ Specify connectivity amongst vertices to form primitives: polygons, quads, triangles.
- ▶ Eliminate redundant transfer and processing of vertex data: coordinates, normals, colours etc.
- ▶ Pass *client space* array pointers in into the OpenGL renderer.
- ▶ OpenGL transfers vertex data using the client space array pointers into *server space* for processing and rendering.
- ▶ (Probably) still requires transfer of data each time mesh is rendered. Some data will change, some will not.
- ▶ Use *vertex buffer objects* to store data in server space, i.e., on the GPU to avoid repeated transfer, and to cache processed data.

Vertex Arrays: Steps

Three steps to use vertex arrays

1. Activate/enable data arrays
2. Specify data
3. Dereference and render.

Step 1: Enable Arrays

```
void glEnableClientState( GLenum cap )
```

is used to specify the array to enable.

The *cap* parameter can take on a symbolic constant value:
GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_EDGE_FLAG_ARRAY,
GL_INDEX_ARRAY, GL_NORMAL_ARRAY,
GL_TEXTURE_COORD_ARRAY, and GL_VERTEX_ARRAY.

Examples:

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

Step 2: Specify Data

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

is used to specify the location and data type and format of coordinate data.

The *pointer* parameter is the client space pointer to the coordinate data. The *size* parameter specifies the number of coordinates per vertex. The *type* parameter can be: `GL_SHORT`, `GL_INT`, `GL_FLOAT`, `GL_DOUBLE`. The *stride* parameter is the offset between consecutive coordinate values, with 0 specifying tight packed.

There are similar calls for the seven other arrays.

Step 3: Dereference and Render

There are three calls for dereferencing data using indices:

1. Single array element

```
void glVertexElement( GLint ith )
```

2. List of array elements

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

3. List of list of array elements

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

Dereference and Render: Single Array Element

Example

```
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
glVertexPointer (2, GL_INT, 0, vertices);
glColorPointer (3, GL_FLOAT, 0, colors);

glBegin (GL_TRIANGLES);
    glArrayElement (2); glArrayElement (3); glArrayElement (5);
glEnd ();
```

has the same effect as

```
glBegin (GL_TRIANGLES);
glColor3fv (colors + (2 * 3) );
glVertex2iv (vertices + (2 * 2) );
glColor3fv (colors + (3 * 3) );
glVertex2iv (vertices + (3 * 2) );
glColor3fv (colors + (5 * 3) );
glVertex2iv (vertices + (5 * 2) );
glEnd ();
```

Dereference and Render: List of Array Elements

The effect of

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

is similar to

```
glBegin (mode);  
    for (i = 0; i < count; i++)  
        glArrayElement (indices[i]);  
glEnd ();
```

Cube glDrawElements() Example

For the cube shown earlier could use `glDrawElements()` as follows:

```
glEnableClientState (GL_VERTEX_ARRAY);  
glVertexPointer (3, GL_FLOAT, 0, vertices);  
  
static GLubyte frontIndices[] = { 4, 5, 6, 7 };  
static GLubyte backIndices[] = { 0, 3, 2, 1 };  
static GLubyte leftIndices[] = { 0, 4, 7, 3 };  
static GLubyte rightIndices[] = { 1, 2, 6, 5 };  
static GLubyte bottomIndices[] = { 0, 1, 5, 4 };  
static GLubyte topIndices[] = { 2, 3, 7, 6 };  
  
glDrawElements (GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);  
glDrawElements (GL_QUADS, 4, GL_UNSIGNED_BYTE, backIndices);  
glDrawElements (GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);  
glDrawElements (GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);  
glDrawElements (GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);  
glDrawElements (GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

Dereference and Render: List of Lists of Array Elements

The effect of

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

is similar to

```
for (i = 0; i < primcount; i++) {  
    if (count[i] > 0)  
        glDrawElements (mode, count[i], type, indices[i]);  
}
```

Cube glMultiDrawElements() Example

For the cube could use glMultiDrawElements() as follows:

```
static GLubyte* allIndices[] = {
    frontIndices,
    backIndices,
    leftIndices,
    rightIndices,
    bottomIndices,
    topIndices
};
static GLsizei allIndicesCount[] = { 4, 4, 4, 4, 4, 4 };

glMultiDrawElements (GL_QUADS, allIndicesCount,
                    GL_UNSIGNED_BYTE, allIndices, 6);
```

Dereferencing and Rendering: Sequence of Array Elements

In the above approaches indices are used to provide indirect addressing into the vertex array and thereby “hop around”.

Another approach provided is to form primitives from the vertices in the vertex array in the sequence they are stored.

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

This has a similar effect to

```
glBegin(mode);  
for (i = 0; i < count; i++) {  
    glVertex (first + i);  
}
```

Interleaved Arrays

Vertex data may be interleaved. That means instead of having separate arrays for say vertex coordinates and vertex colours a single array is used with the coordinate and colour values alternating.

Data from interleaved arrays may be extracted by specifying an appropriate *stride* value in the `glVertexPointer()`, `glColorPointer()` and other similar pointer calls.

`glInterleavedArrays()` allows accessing interleaved data in a single call:

```
void glInterleavedArrays(  GLenum format,
                          GLsizei stride,
                          void *pointer)
```

Initialises all eight arrays and disables unused arrays according to the GLenum *format* parameter. *stride* indicates the number of bytes between vertices. *pointer* is a pointer to the data array.

Vertex Buffer Objects: VBOs

- ▶ Vertex arrays are stored in client space.
- ▶ Vertex arrays in client space reduce redundant vertex data.
- ▶ Still need to transfer vertex data into server space, usually repeatedly.
- ▶ *Vertex buffer objects* or VBOs allow storing of vertex arrays in server space

Vertex Buffer Objects: Steps

Six steps to use vertex buffer objects

1. Generate buffer object identifiers
2. Bind a buffer object, specifying for vertex data or indices
3. Request storage, optionally initialise
4. Specify data including offsets into buffer object
5. Bind buffer object to be used in rendering
6. Render using vertex array techniques, e.g. `glDrawElements`

Step 1: Create Buffer Objects

- ▶ Similar to creating identifiers for display lists using `glGenLists`
- ▶ To allocate one or more buffer object identifiers use
`void glGenBuffers(GLsizei n, GLuint *buffers)`
- ▶ `glIsBuffer(GLuint buffer)` to test if an integer is in use
- ▶ Zero is reserved identifier
- ▶ To delete use `glDeleteBuffers`

Step 2: Bind Buffer Object

- ▶ Binding makes a buffer object active
- ▶ Once bound a buffer object is used for operations to initialise it with data and/or for vertex array rendering operations
- ▶ Use

```
void glBindBuffer(GLenum target, GLuint *buffer)
```
- ▶ *target* can be `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`

Step 3: Allocate and Initialise

- ▶ Need to reserve space for the buffer object in the OpenGL server
- ▶ Once bound a buffer object is used for operations to initialise it with data and/or for vertex array rendering operations
- ▶ Use

```
void glBufferData(  GLenum target, GLsizei size,
                    const GLvoid *data, GLenum usage)
```
- ▶ *target* can be `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`
- ▶ *size* is number of bytes
- ▶ *data* is pointer to client memory or `NULL`
- ▶ *usage* is a hint for performance
- ▶ Memory is finite, may get `GL_OUT_OF_MEMORY`
- ▶ Store vertex, color, normal etc data in the one VBO

Step 4: Specify Data

- ▶ Earlier we saw `glVertexPointer`, `glNormalPointer`, `glColorPointer` etc for specifying where the data for vertex arrays is to be found (on the client)
- ▶ The same functions must be used for vertex buffer objects
- ▶ Except the data is in the vertex buffer object, initialised with `glBufferData` as above
- ▶ Can be interleaved or non-interleaved
- ▶ The *pointer* argument becomes an *offset* in the VBO
- ▶ If data is stored in an interleaved array *stride* is non-zero

Step 5: Bind

- ▶ Use `glBindBuffer` as before
- ▶ No need to rebind if buffer already bound from step 2

Step 6: Render

- ▶ Use `glDrawArrays`, `glDrawElements` etc as for vertex arrays