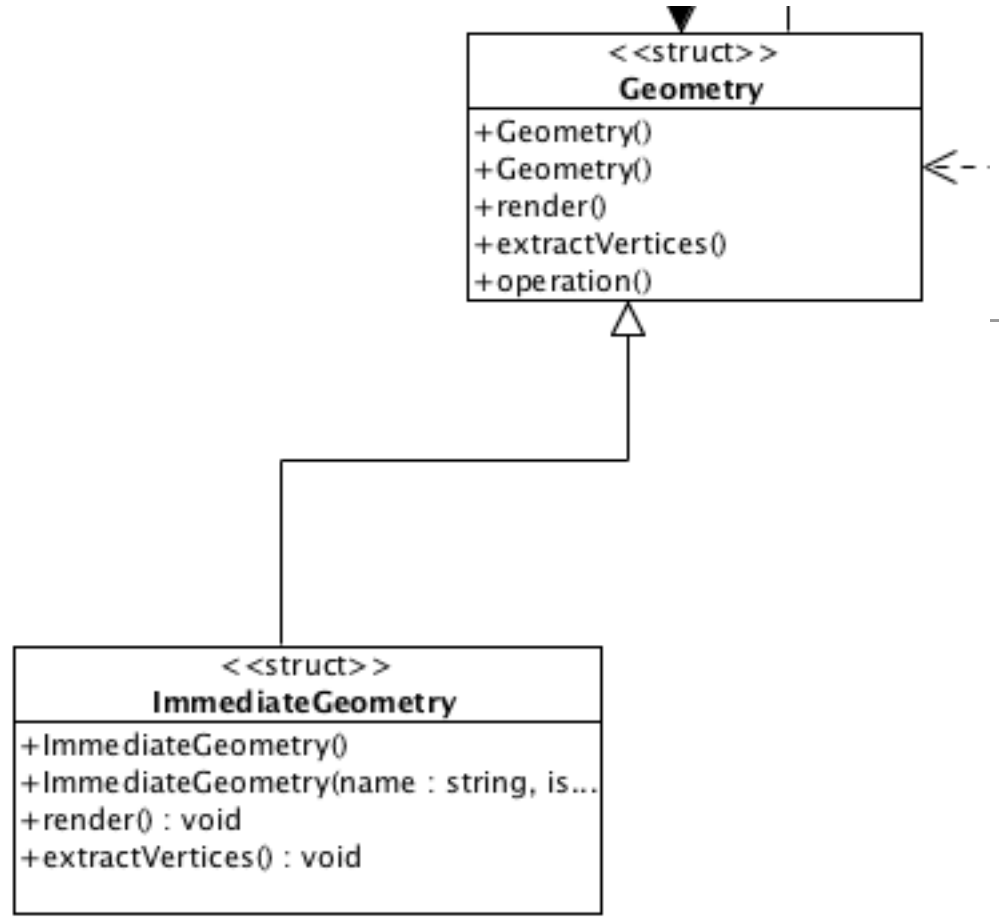
VBO

Refactor Model



- Make Geometry abstract
- Incorporate immediate mode current rendering into derived class

```
struct Geometry : public Renderable
{
  std::string name;
  std::vector<Face> faces;
  VertexGroup *vertexGroup;
  std::vector<Vector3> vertices;

  Geometry();
  Geometry(std::string name, std::istream&, VertexGroup*);

  virtual void render()=0;
  virtual void extractVertices()=0;
};
```

```
struct ImmediateGeometry : public Geometry
{
  ImmediateGeometry();
  ImmediateGeometry(std::string    name,
                        std::istream&, VertexGroup*);

  void render();
  void extractVertices();
};
```

ImmediateMode extractVertices()

```
void ImmediateGeometry::extractVertices()
{
  int newIndex=0;
  foreach (Face &face, faces)
  {
    vector<int> groupIndices = face.vertexIndices;
    face.vertexIndices.clear();
    foreach (int index, groupIndices)
    {
      Vector3 vertex(vertexGroup->vertices[index-1].X,
                     vertexGroup->vertices[index-1].Y,
                     vertexGroup->vertices[index-1].Z);
      vertices.push_back(vertex);
      face.vertexIndices.push_back(newIndex);
      newIndex++;
    }
  }
}
```

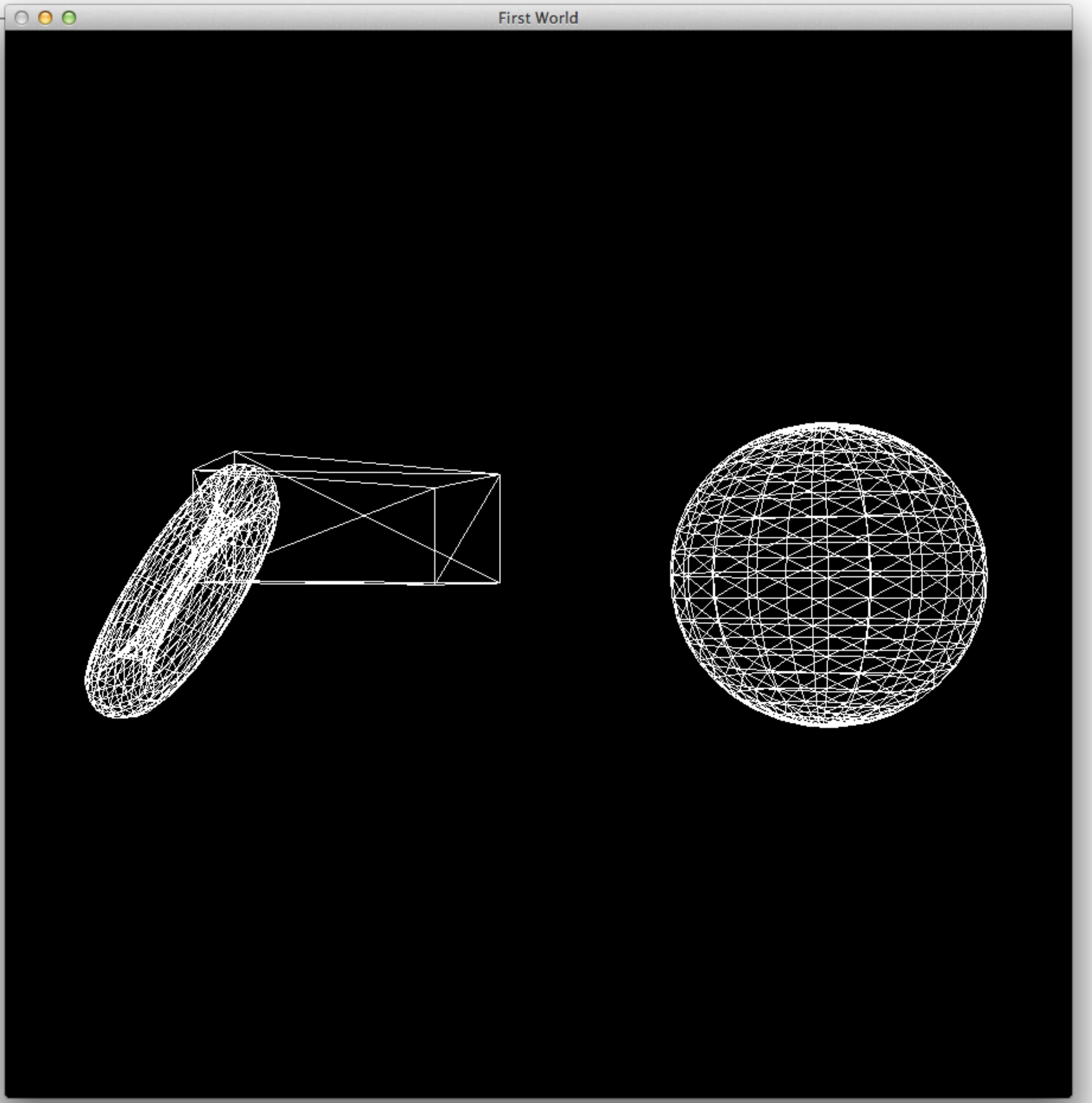
render()

```
void ImmediateGeometry::render()
{
  foreach (Face &face, faces)
  {
    face.render(vertices);
  }
}
```

```
void Face::render(std::vector <Vector3>&vertexTable)
{
  vertexIndices.size() == 3?
     glBegin(GL_TRIANGLES)
    :glBegin(GL_QUADS);

  foreach (int index, vertexIndices)
  {
    glVertex3f( vertexTable[index].X,
                vertexTable[index].Y,
                vertexTable[index].Z );
  }
  glEnd();
}
```

Metrics - Immediate

- **Immediate**
  - **Vertices for pCube1: 36**
  - **Vertices for pSphere1: 2280**
  - **Vertices for pTorus1: 2400**



Immediate Mode

- So far, all of our primitive batches have been assembled using glBegin/glEnd pairs with individual glVertex calls between them.
- This is a very flexible means of assembling a batch of primitives, and is easy to use and understand.
- Unfortunately, when performance is taken into account, it is also the worst possible way to submit geometry to graphics hardware.

Example

- For a single triangle, that’s 6 function calls. Each of these functions contains potentially expensive validation code in the OpenGL driver.

- In addition, we must pass 18 different four byte parameters pushed on the stack, and of course return to the calling function - for a single triangle!

- Now, multiply this by a 3D scene containing 10,000 or more triangles, and it is easy to imagine the graphics hardware sitting idle waiting on the CPU to assemble and submit geometry batches.

```
glBegin(GL_TRIANGLES);
  glNormal3f(x, y, z);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glVertex3f(x, y, z);
glEnd();
```

Amelioration Strategies

- Use vector-based functions such glVertex3fv to consolidate calls
- Use strips and fans to reduce redundant transformations and copies.
- However, these approaches are flawed from a performance standpoint because they require many thousands of very small, potentially expensive operations to submit a batch of geometry.
- This method of submitting geometry batches is often called immediate mode rendering and is largely deprecated in modern OpenGL development

Alternative 1: Display Lists

- Most OpenGL rendering commands are converted into some hardware-specific commands.
- There commands are not dispatched immediately to the hardware. Instead, they are accumulated in a local buffer until some threshold is reached, at which point they are flushed to the hardware.
- Often, the geometry or other OpenGL data remains the same from frame to frame.
- OpenGL provides a facility to create a preprocessed set of OpenGL commands can then be quickly copied to the command buffer for more rapid execution.
- This precompiled list of commands is called a display list

Creating Display Lists

- Delimit a display list with glNewList/glEndList.
- The named display list now contains all OpenGL rendering commands that occur between
- the glNewList and glEndList function calls. The GL_COMPILE parameter tells OpenGL to compile the list but not to execute it
- A display list, containing any number of precompiled OpenGL commands, is then executed with a single command:

```
glNewList(<unsigned integer name>,GL_COMPILE);
...
...
// Some OpenGL Code
...
...
glEndList();
```

```
void glCallList(GLuint list);
```

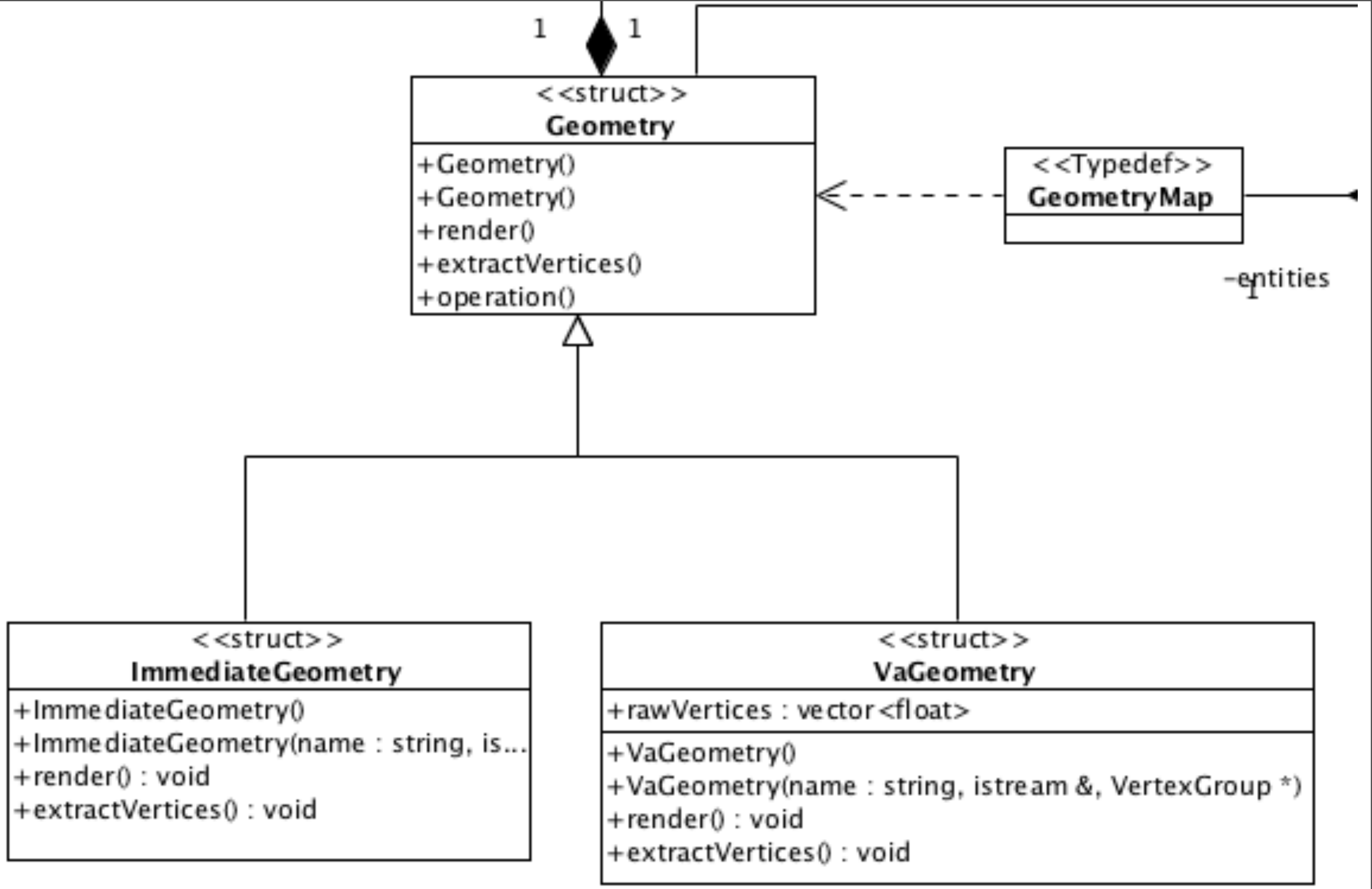
Display List Caveat

- Display lists are useful for precompiled lists of static geometry
- Display lists cannot contain calls that create display lists.
- Geometry cannot change, or else now display list must be generated.

Alternative 2: Vertex Arrays

- Vertex arrays can be almost as fast as display lists, but without the requirement that the geometry be static.
- Four steps:

1. Assemble geometry data in one or more arrays.
2. Tell OpenGL where the data is. When rendering is performed, OpenGL pulls the vertex data directly from the arrays you have specified.
3. Explicitly tell OpenGL which arrays you are using. You can have separate arrays for vertices, normals, colors, and so on, and you must let OpenGL know which of these data sets you want to use.
4. Execute the OpenGL commands to actually perform the rendering using your vertex data.



```
struct VaGeometry : public Geometry
{
  std::vector<float> rawVertices;

  VaGeometry();
  VaGeometry(std::string name, std::istream&, VertexGroup*);

  void render();
  void extractVertices();
};
```

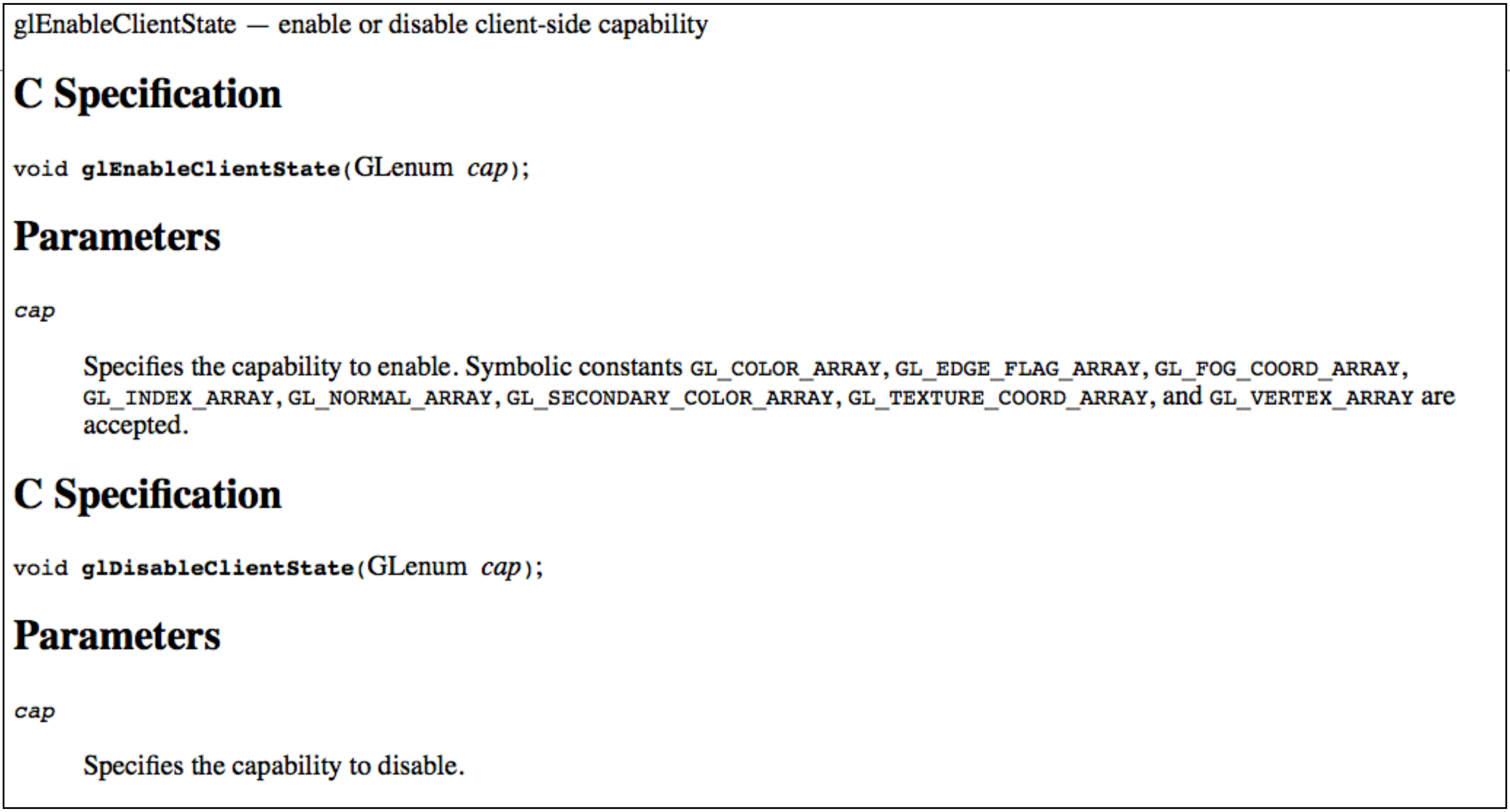
VaGeometry::extractVertices()

- Iterate through all faces
- Copy all the vertices into a simple vertex array called “rawVertices”

```
void VaGeometry::extractVertices()
{
  foreach (Face &face, faces)
  {
    foreach (int index, face.vertexIndices)
    {
      Vector3 vertex(vertexGroup->vertices[index-1].X,
                     vertexGroup->vertices[index-1].Y,
                     vertexGroup->vertices[index-1].Z);
      rawVertices.push_back(vertex.X);
      rawVertices.push_back(vertex.Y);
      rawVertices.push_back(vertex.Z);
    }
  }
}
```

Rendering Vertex Arrays

- void glEnableClientState(GLenum cap);
- void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);
- void glDrawArrays(GLenum mode, GLint first, GLsizei count);
- void glDisableClientState(GLenum cap);



glEnableClientState — enable or disable client-side capability

C Specification

`void glEnableClientState(GLenum cap);`

Parameters

cap

Specifies the capability to enable. Symbolic constants `GL_COLOR_ARRAY`, `GL_EDGE_FLAG_ARRAY`, `GL_FOG_COORD_ARRAY`, `GL_INDEX_ARRAY`, `GL_NORMAL_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, and `GL_VERTEX_ARRAY` are accepted.

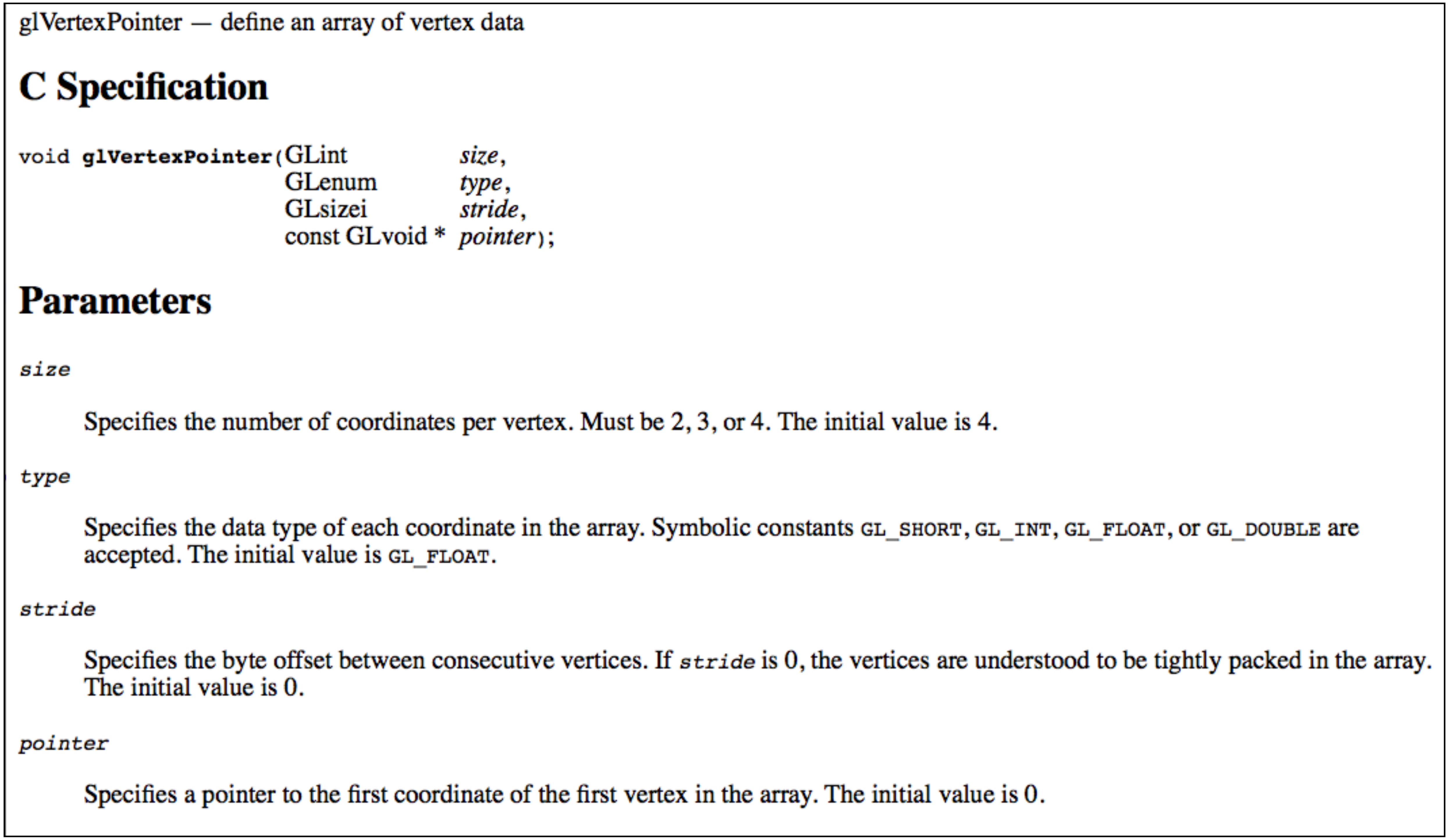
C Specification

`void glDisableClientState(GLenum cap);`

Parameters

cap

Specifies the capability to disable.



glVertexPointer — define an array of vertex data

C Specification

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

Parameters

size

Specifies the number of coordinates per vertex. Must be 2, 3, or 4. The initial value is 4.

type

Specifies the data type of each coordinate in the array. Symbolic constants `GL_SHORT`, `GL_INT`, `GL_FLOAT`, or `GL_DOUBLE` are accepted. The initial value is `GL_FLOAT`.

stride

Specifies the byte offset between consecutive vertices. If *stride* is 0, the vertices are understood to be tightly packed in the array. The initial value is 0.

pointer

Specifies a pointer to the first coordinate of the first vertex in the array. The initial value is 0.

glDrawArrays — render primitives from array data

C Specification

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

Parameters

mode

Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON` are accepted.

first

Specifies the starting index in the enabled arrays.

count

Specifies the number of indices to be rendered.

VaGeometry::render()

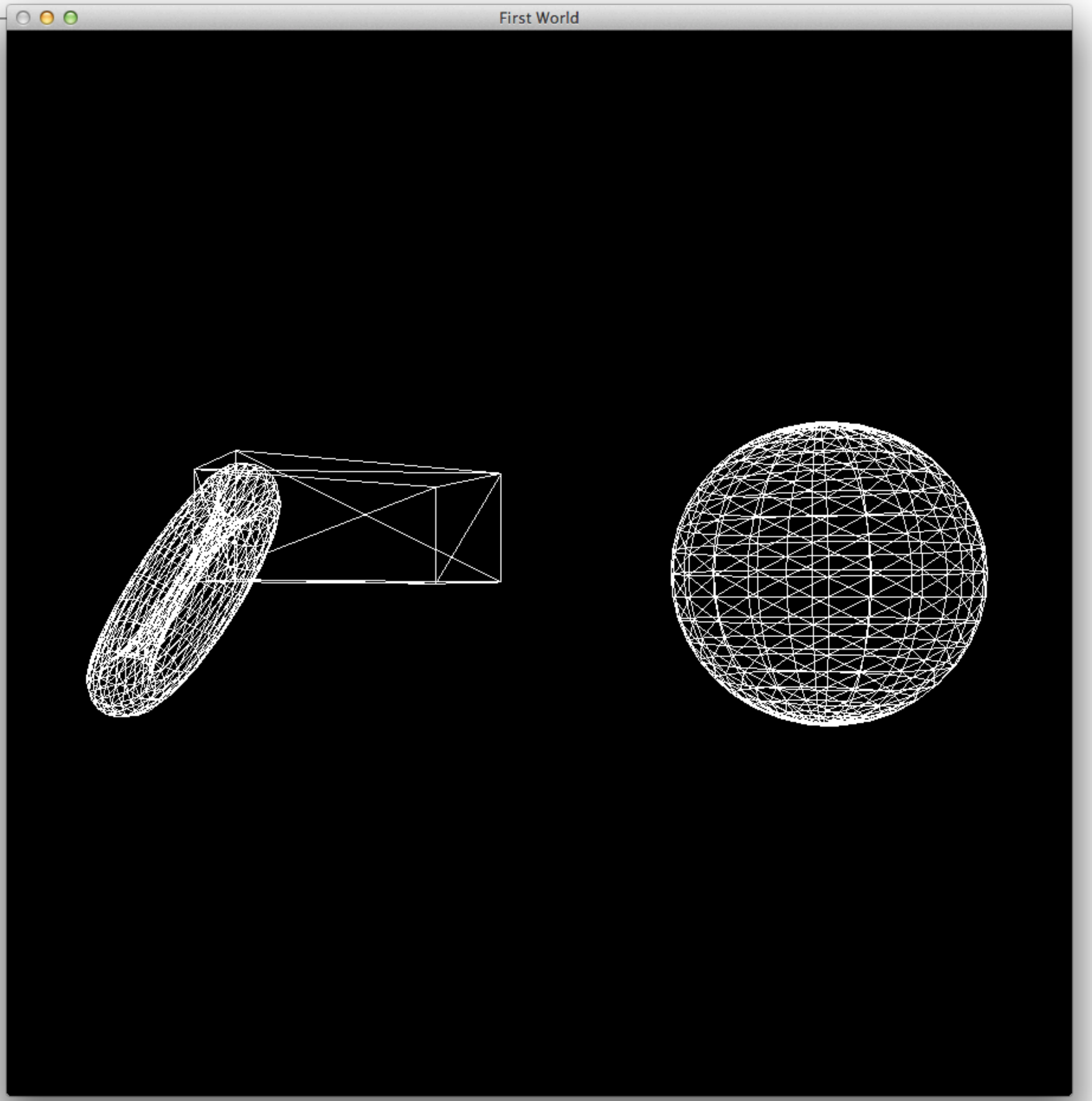
- Face is no longer needed
- Computing the address of an element in an stl::vector is compatible with a float array.
- Geometry must be ‘Triangulated’

```
void VaGeometry::render()
{
  glEnableClientState(GL_VERTEX_ARRAY);

  glVertexPointer(3, GL_FLOAT, 0, &rawVertices[0]);
  glDrawArrays(GL_TRIANGLES, 0, rawVertices.size()/3);

  glDisableClientState(GL_VERTEX_ARRAY);
}
```

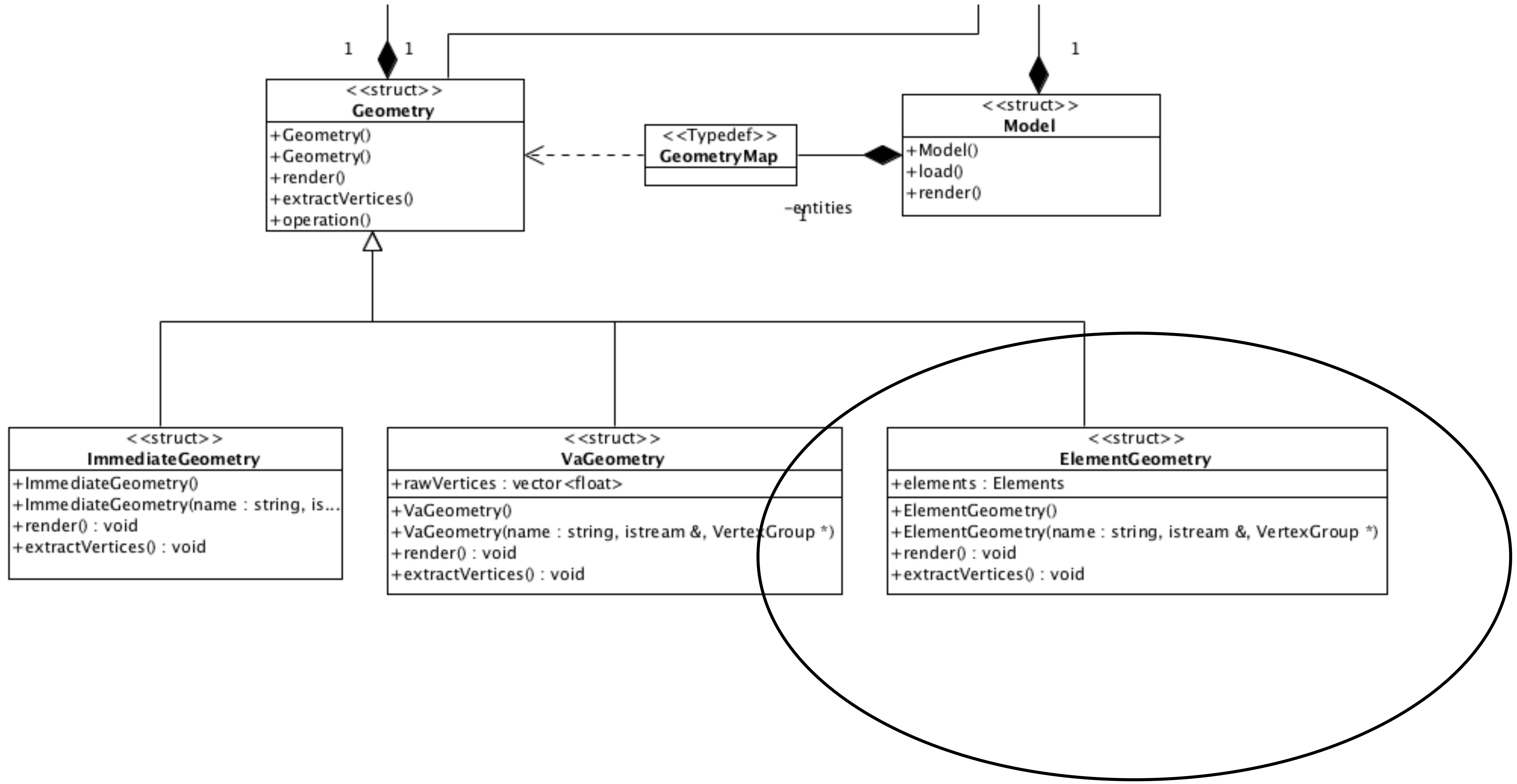
Metrics

- Immediate
  - Vertices for pCube1: 36
  - Vertices for pSphere1: 2280
  - Vertices for pTorus1: 2400

- **VertexArray**
  - **Vertices for pCube1: 108**
  - **Vertices for pSphere1: 6840**
  - **Vertices for pTorus1: 7200**

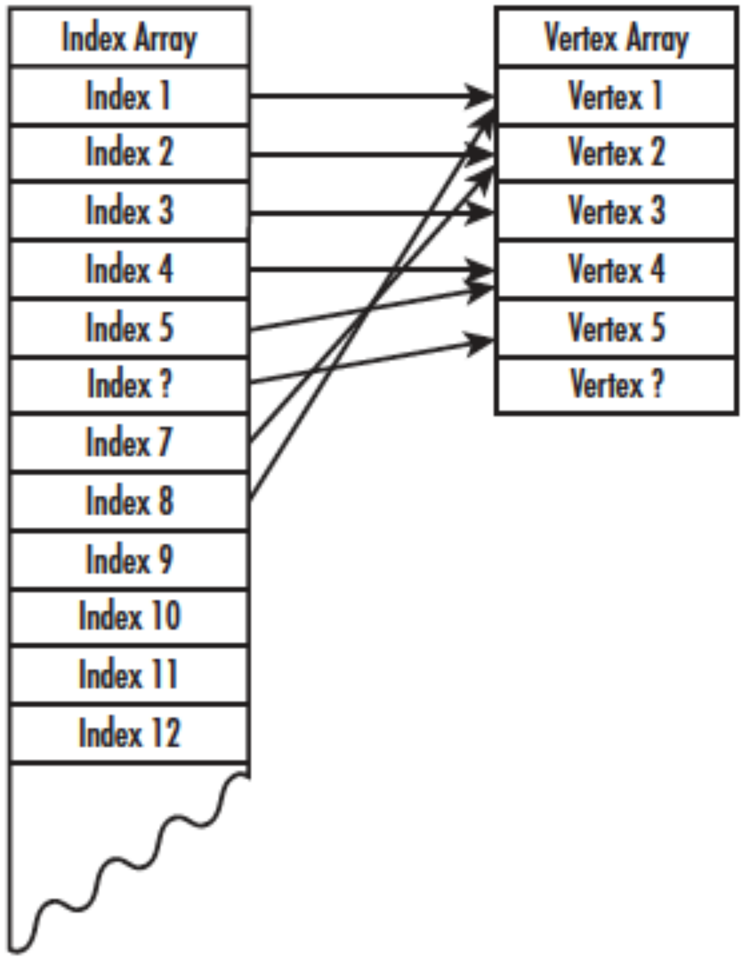


Alternative 2: Indexed Vertex Arrays



Indexed Geometry

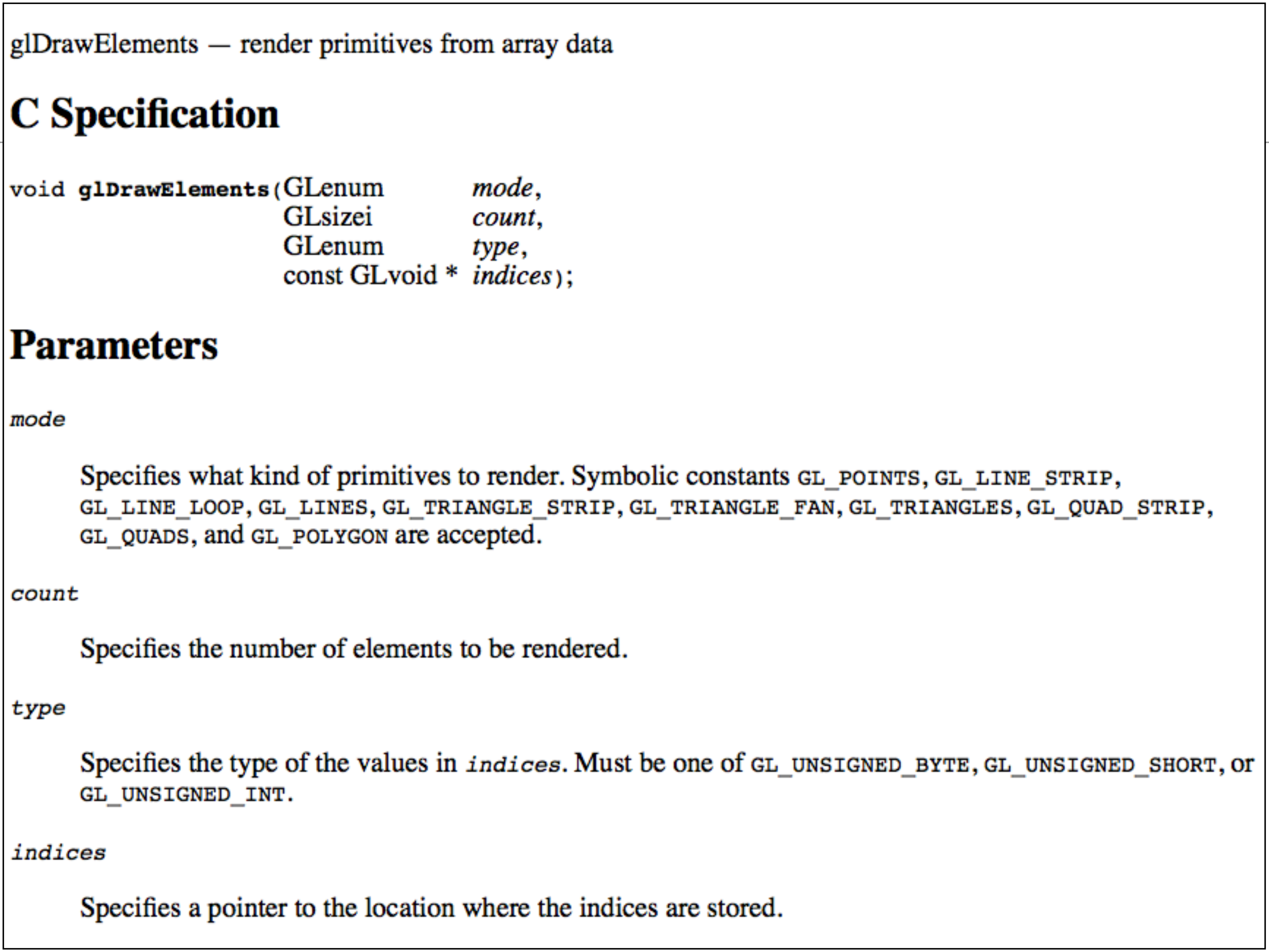
```
struct Elements
{
  std::vector<float>      vertices;
  std::vector<GLushort>   indices;
};

struct ElementGeometry : public Geometry
{
  Elements elements;

  ElementGeometry();
  ElementGeometry(std::string     name,
                  std::istream&, VertexGroup*);

  void render();
  void extractVertices();
};
```



ElementGeometry::extractVertices()

- Effectively remapping the geometry from the vertexGroup into the elements data structure

```
void ElementGeometry::extractVertices()
{
  typedef map <GLushort,GLushort> IndexMap;
  IndexMap indexMap;
  int newIndex=0;
  foreach (Face &face, faces)
  {
    foreach (int index, face.vertexIndices)
    {
      Vector3 vertex(vertexGroup->vertices[index-1].X,
                     vertexGroup->vertices[index-1].Y,
                     vertexGroup->vertices[index-1].Z);
      if (indexMap.find(index) == indexMap.end())
      {
        indexMap[index] = newIndex;
        elements.vertices.push_back(vertex.X);
        elements.vertices.push_back(vertex.Y);
        elements.vertices.push_back(vertex.Z);
        elements.indices.push_back(newIndex);
        newIndex++;
      }
      else
      {
        elements.indices.push_back(indexMap[index]);
      }
    }
  }
}
```



glDrawElements — render primitives from array data

C Specification

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

Parameters

`mode`

Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON` are accepted.

`count`

Specifies the number of elements to be rendered.

`type`

Specifies the type of the values in *`indices`*. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.

`indices`

Specifies a pointer to the location where the indices are stored.

ElementGeometry::render()

- Indicate to OpenGL the vertex array
- then indicate we wish to draw triangles using these vertices based on a set of indices

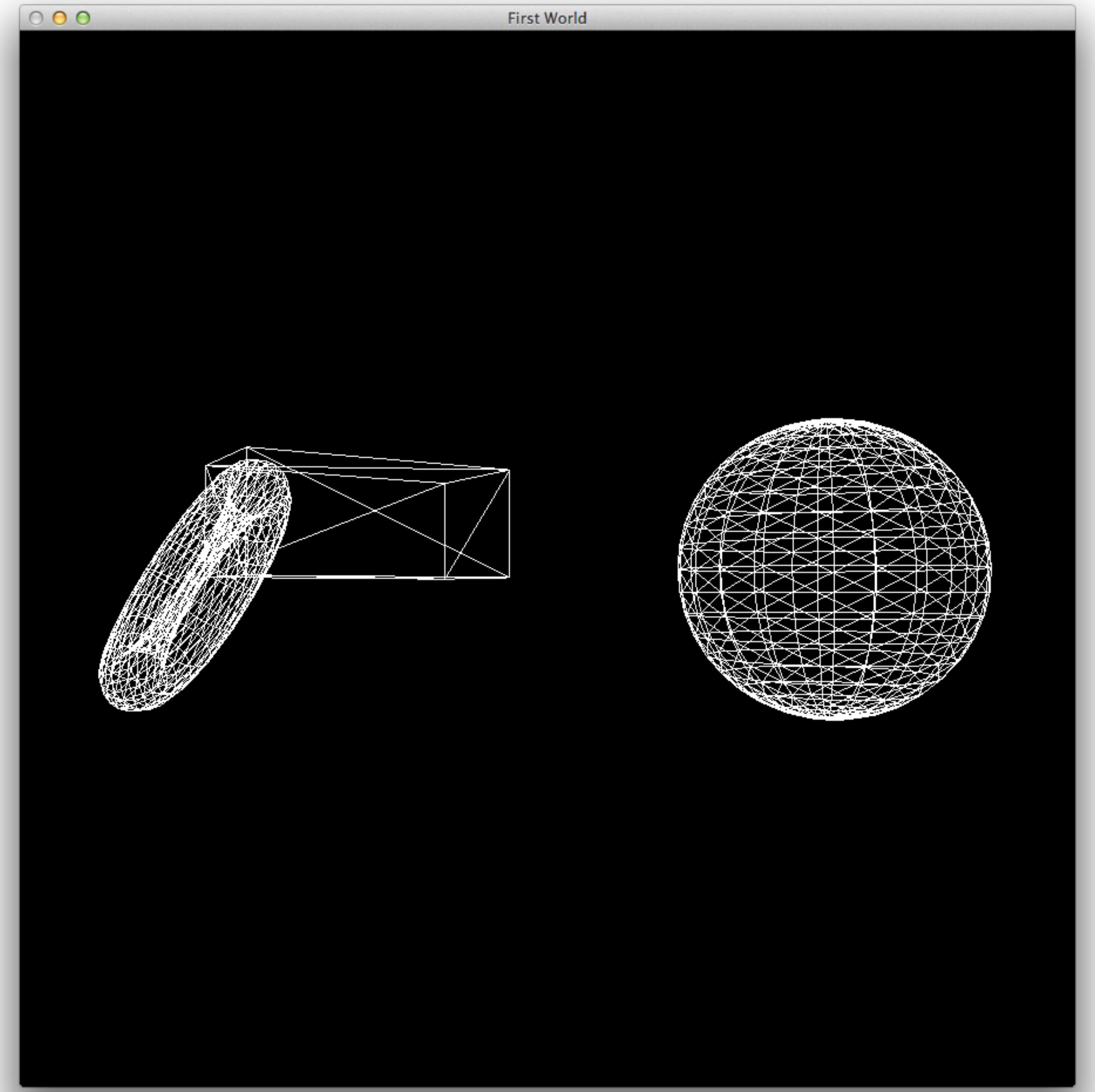
```
void ElementGeometry::render()
{
  glEnableClientState(GL_VERTEX_ARRAY);

  glVertexPointer(3, GL_FLOAT, 0, &elements.vertices[0]);
  glDrawElements(GL_TRIANGLES, elements.indices.size(), GL_UNSIGNED_SHORT, &elements.indices[0]);

  glDisableClientState(GL_VERTEX_ARRAY);
}
```

Metrics

- Immediate
  - Vertices for pCube1: 36
  - Vertices for pSphere1: 2280
  - Vertices for pTorus1: 2400

- VertexArray
  - Vertices for pCube1: 108
  - Vertices for pSphere1: 6840
  - Vertices for pTorus1: 7200

- **Elements**
  - **Vertices for pCube1:24**
  - **Vertices for pSphere1:1146**
  - **Vertices for pTorus1:1200**



Review: Display Lists

- Display lists are a quick and easy way to optimize immediate mode code .
- At worst, a display list will contain a precompiled set of OpenGL data, ready to be copied quickly to the command buffer, and destined for the graphics hardware.
- At best, an implementation may copy a display list to the graphics hardware, reducing bandwidth to the hardware to essentially nil.

Review: Vertex Arrays

- At worst result in block copies (still much faster than immediate mode) to the hardware.

- Indexed vertex arrays (ElementGeometry) further up the ante by providing a means of reducing the amount of vertex data that must be transferred to the hardware, and reducing the transformation overhead.

- Vertex Array
  - Vertices for pCube1: 108
  - Vertices for pSphere1: 6840
  - Vertices for pTorus1: 7200

- Elements (Indexed Vertex Array)
  - Vertices for pCube1:24
  - Vertices for pSphere1:1146
  - Vertices for pTorus1:1200

Vertex Buffer Objects

Ability to transfer individual arrays from your client (CPU-accessible) memory to the graphics hardware. Steps are:

- Use vertex arrays (as we have been doing)
- Create the buffer objects using glGenBuffers:

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

- The first parameter is the number of buffer objects desired, and the second is an array that is filled with new vertex buffer object names
- Vertex buffer objects are “bound” using

```
void glBindBuffer(GLenum target, GLuint buffer);
```

- target refers to the kind of array being bound.This may be either GL_ARRAY_BUFFER for vertex data (including normals, texture coordinates, etc.) or GL_ELEMENT_ARRAY_BUFFER for array indexes to be used with glDrawElements and the other index-based rendering functions.

Loading the VBO

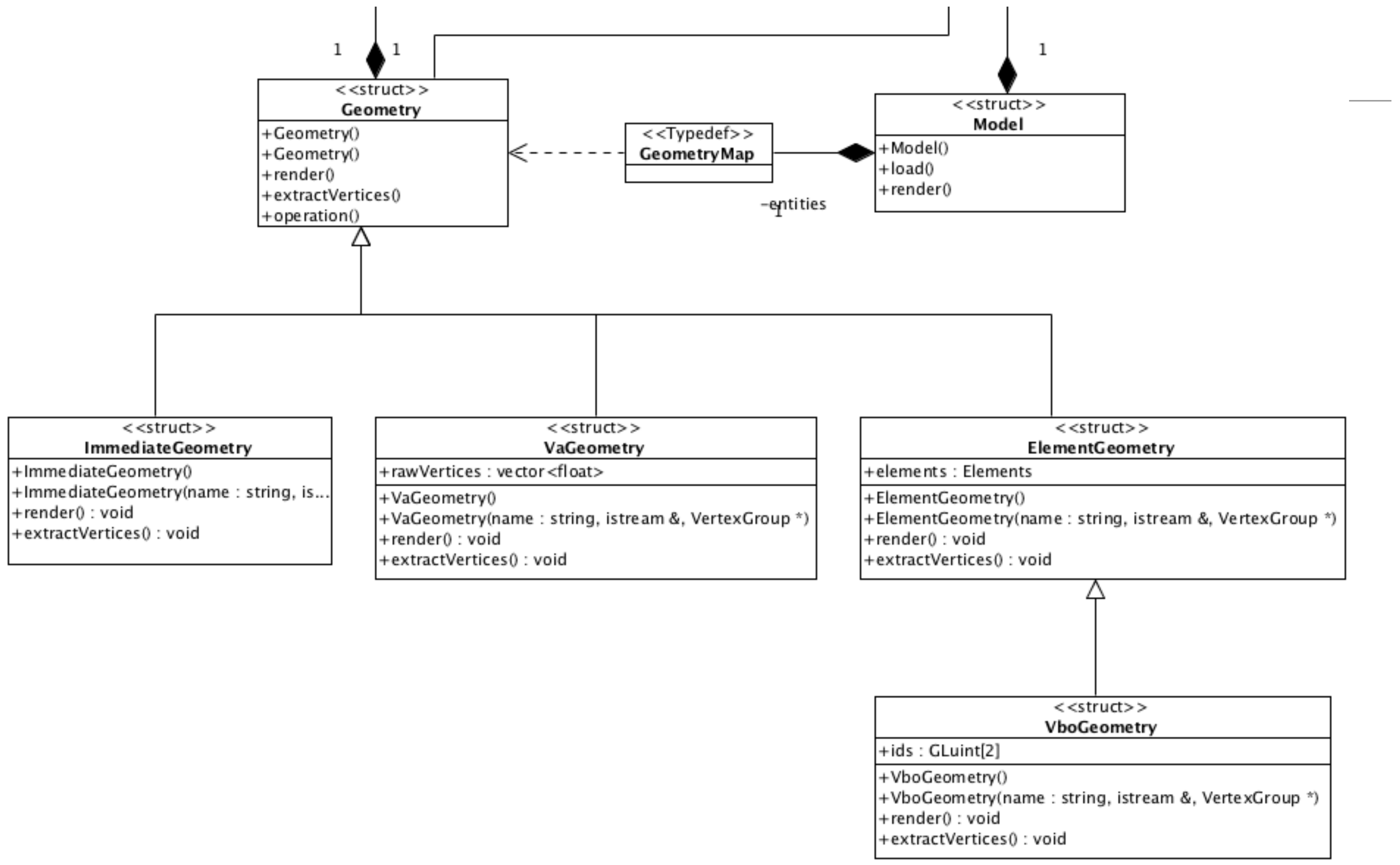
- To copy vertex data to the graphics hardware, first bind to the buffer object in question, then call glBufferData:

  ```
  void glBufferData(GLenum target, GLsizeiptr size, GLvoid *data, GLenum usage);
  ```

- Again target refers to either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER, and size refers to the size in bytes of the vertex array.

void glBufferData(GLenum target, GLsizeiptr size, GLvoid *data, GLenum usage);

- GL_DYNAMIC_DRAW The data stored in the buffer object is likely to change frequently but is likely to be used as a source for drawing several times in between changes. This hint tells the implementation to put the data somewhere it won’t be too painful to update once in a while.
- GL_STATIC_DRAW The data stored in the buffer object is unlikely to change and will be used possibly many times as a source for drawing. This hint tells the implementation to put the data somewhere it’s quick to draw from, but probably not quick to update.
- GL_STREAM_DRAW The data store in the buffer object is likely to change frequently and will be used only once (or at least very few times) in between changes. This hint tells the implementation that you have time-sensitive data such as animated geometry that will be used once and then replaced. It is crucial that the data be placed somewhere quick to update, even at the expense of faster rendering.


1
1
1
<<struct>>
Geometry
+Geometry()
+Geometry()
+render()
+extractVertices()
+operation()
<<Typedef>>
GeometryMap
-entities
1
<<struct>>
Model
+Model()
+load()
+render()
<<struct>>
ImmediateGeometry
+ImmediateGeometry()
+ImmediateGeometry(name : string, is...
+render() : void
+extractVertices() : void
<<struct>>
VaGeometry
+rawVertices : vector<float>
+VaGeometry()
+VaGeometry(name : string, istream &, VertexGroup *)
+render() : void
+extractVertices() : void
<<struct>>
ElementGeometry
+elements : Elements
+ElementGeometry()
+ElementGeometry(name : string, istream &, VertexGroup *)
+render() : void
+extractVertices() : void
<<struct>>
VboGeometry
+ids : GLuint[2]
+VboGeometry()
+VboGeometry(name : string, istream &, VertexGroup *)
+render() : void
+extractVertices() : void

VboGeometry

- Derived from ElementGeometry
- Incorporate ids for 2 buffers
  - ids[0] - indices
  - ids[1] - vertices

```
struct VboGeometry : public ElementGeometry
{
  GLuint ids[2];

  VboGeometry();
  VboGeometry(std::string    name,
              std::istream&, VertexGroup*);

  void render();
  void extractVertices();
};
```

VboGeometry::extractVertices

```
void VboGeometry::extractVertices()
{
  ElementGeometry::extractVertices();

  glGenBuffers(2, ids);

  glBindBuffer(GL_ARRAY_BUFFER, ids[0]);
  glBufferData(GL_ARRAY_BUFFER, elements.vertices.size()*sizeof(float), &elements.vertices[0], GL_STATIC_DRAW);

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ids[1]);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, elements.indices.size()*sizeof(GLushort), &elements.indices[0], GL_STATIC_DRAW);
}
```

- Call base class to extract vertices
- Generate 2 new buffers - and bind them to the vertices and indixes
- Use “STATIC” usage hint, indicating that the vertices wont be changed

VboGemoetry::render()

```
void VboGeometry::render()
{
  glBindBuffer(GL_ARRAY_BUFFER, ids[0]);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ids[1]);

  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(3, GL_FLOAT, 0, 0);
  glDrawElements(GL_TRIANGLES, elements.indices.size(), GL_UNSIGNED_SHORT, 0);
  glDisableClientState(GL_VERTEX_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER, 0);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

- Bind the index and vertex buffers
- Draw the triangles as specified by the index buffer
- Unbind the buffer

Updating the Buffer

- If the buffer is to be updated - and we have disposed of the original data (not in our case), we can map the buffer back into client memory:

  ```
  void * glMapBuffer(GLenum  target,  GLenum  access);
  ```

- glMapBuffer maps to the client's address space the entire data store of the buffer object currently bound to target.
- The data can then be directly read and/or written relative to the returned pointer, depending on the specified access policy.

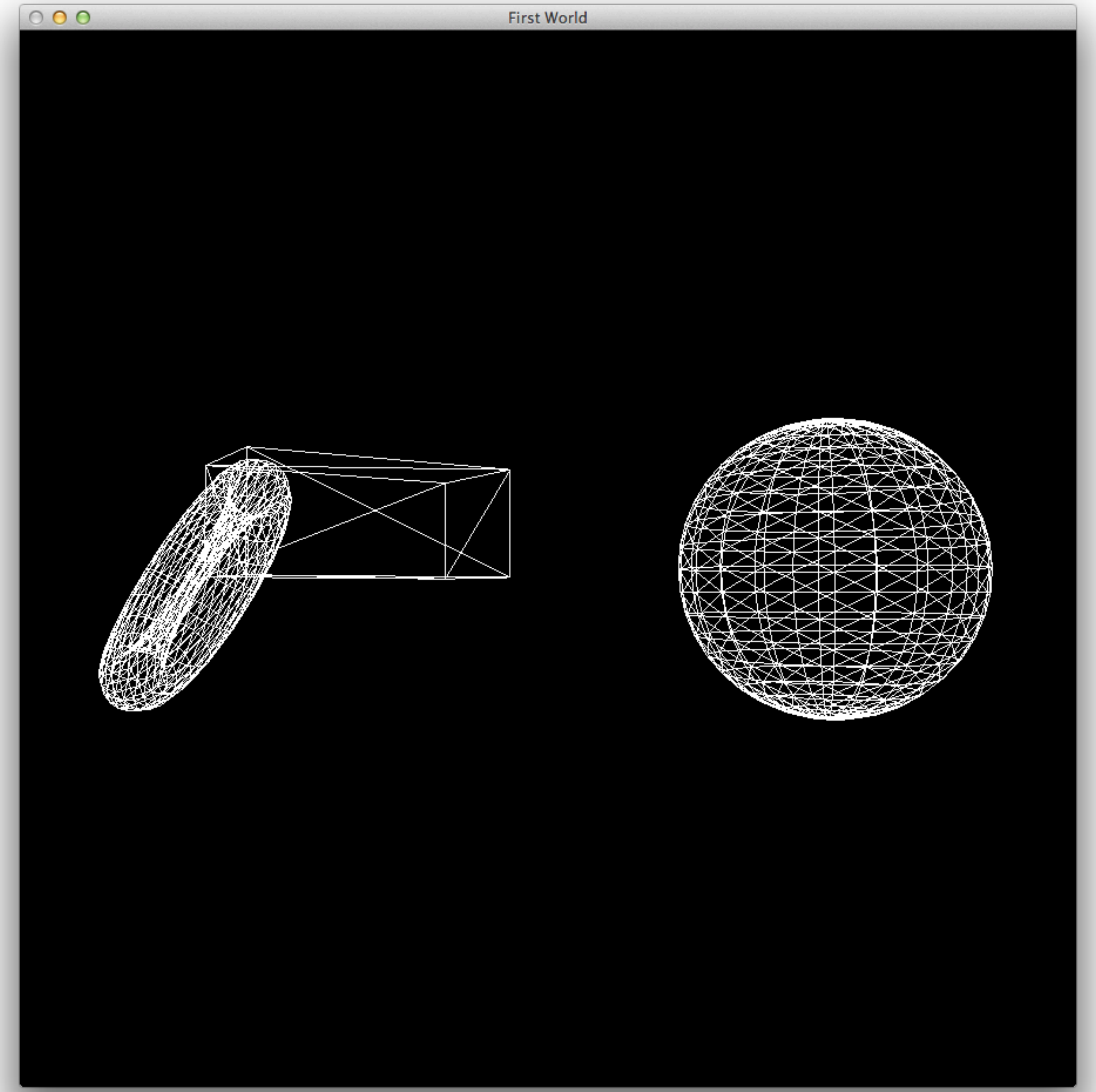
Metrics

- Immediate
  - Vertices for pCube1: 36
  - Vertices for pSphere1: 2280
  - Vertices for pTorus1: 2400

- VertexArray
  - Vertices for pCube1: 108
  - Vertices for pSphere1: 6840
  - Vertices for pTorus1: 7200

- Elements
  - Vertices for pCube1:24
  - Vertices for pSphere1:1146
  - Vertices for pTorus1:1200

- **VBO**
  - **Vertices for pCube1:24**
  - **Vertices for pSphere1:1146**
  - **Vertices for pTorus1:1200**


First World