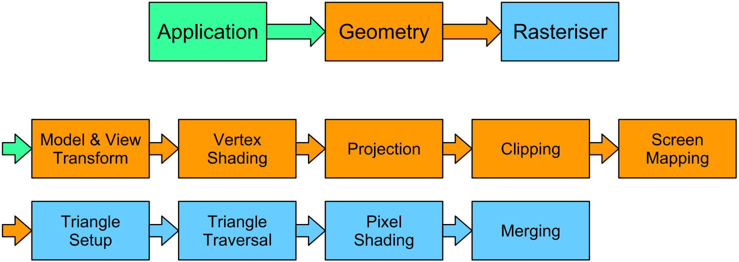
159.709 Computer Graphics

Lecture 02 - OpenGL Pipeline

Daniel Playne
d.p.playne@massey.ac.nz

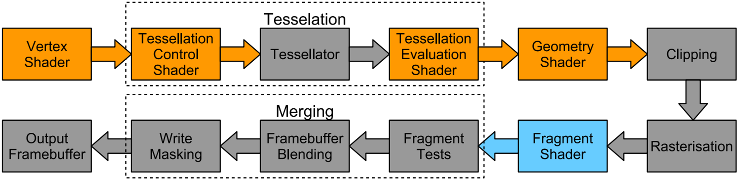
Rendering Pipeline Overview

Reminder: the three main stages (and substages) of the rendering graphics pipeline.



Rendering Pipeline Overview

The rendering pipeline is implemented in OpenGL with the following set of stages.



The programmable stages are coloured by the stage they belong to while the non-programmable (they may still be configurable) are shown in grey.

Rendering Pipeline Overview

The ***vertex*** and ***fragment*** shaders are necessary parts of the pipeline, these define how vertexes and subsequent fragments should be processed. We will look at these in some detail today.

The ***tesselation*** and ***geometry*** shaders represent an optional part of the pipeline that can be used to generate new vertexes and/or eliminate/modify existing vertexes. These will be covered later in the course.

OpenGL Shaders

The programmable stages of the OpenGL pipeline are called ***shaders*** and are written in a language called OpenGL Shading Language or GLSL.

- First introduced in OpenGL 2.0
- GLSL 1.5.0 was released alongside OpenGL 3.2 (oldest version we will target)
- Version numbers have been aligned since OpenGL 3.3 (GLSL 3.30).
- GLSL shaders are compiled at runtime and linked into program objects that can be executed on the GPU.

OpenGL Shaders

OpenGL ***shaders*** are essentially little programs that are executed on the graphics card. These shader programs are executed in parallel using a programming model called SIMT or Single Instruction Multiple Thread. For our purposes this is the same as SIMD or Single Instruction Multiple Data - the same shader program is run many times but will be given different data as input.

For example, the same vertex shader will be invoked many different times with each invocation being given a different vertex to process.

Vertex Shader

The ***vertex shader*** is the first stage of the OpenGL pipeline that does any graphical processing (some automatic processing does actually take place before this).

When a set of primitives (triangles, lines, points etc) is drawn, one invocation of the ***vertex shader*** will be generated for each vertex. Each invocation is responsible for processing that vertex and determining what information to pass along the pipeline.

Vertex Shader

The functions the ***vertex shader*** will perform may include (but are not necessarily limited to):

- Apply ***model***, ***view*** and ***perspective*** transformations.
- Write the transformed coordinates to `gl_Position`.
- Transform associated vertex data (colour, normal, texture coordinates etc).
- Procedural deformations and/or animations.

Vertex Shader

There are some restrictions on what the vertex shader can access and what it can produce:

- Cannot create or destroy vertexes.
- Cannot communicate with any other vertex shader.
- Does not have access to any other vertexes in the same triangle (or other primitive).

Vertex Shader

The ***vertex shader*** outputs the position of the vertex (written to `gl_Position`) in clip coordinates which are used by OpenGL to clip primitives outside the viewport.

The clip coordinates are automatically converted to Normalised Device Coordinates (NDC).

In addition to this required position information, the ***vertex shader*** can also pass other vertex information along to the ***fragment shader***. This information can include - colours, normals, texture coordinates and more.

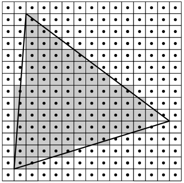
Fragment Shader

The ***fragment shader*** is invoked once for each ***fragment*** that is generated by the rasteriser. The main job of the ***fragment*** is to determine the colour of the fragment.

This shader is sometimes referred to as the ***pixel shader*** which can be somewhat misleading. There may be multiple fragments (from different primitives) that all cover the same pixel. If the scene involves partially transparent objects or fragments only partially cover a pixel then the colour value from multiple fragments may contribute to the final pixel value.

Fragment Shader

The fragments are generated automatically by the OpenGL pipeline rasteriser. Most commonly this involves ‘filling in’ the areas of the triangles using the positions generated by the ***vertex shader***.



This stage can be configured to draw only points or lines between vertexes to draw a wireframe of a model.

Fragment Shader

The input to the ***fragment shader*** is the vertex data output from the ***vertex shader***, this can be defined by the programmer.

If the fragment is at the exact location of one vertex, then the values the fragment receives as input will be the values output by the corresponding vertex shader. Most fragments will not be co-located with a vertex and will be somewhere in the centre of the triangle. In this case the input to the fragment will be linearly interpolated from the output of the relevant vertexes.

Fragment Shader

The main function of the ***fragment shaders*** is to calculate the output colour of a pixel. In most cases this involves performing lighting, shading and texturing operations. However, many different visual effects can be created by the fragment shaders.

Fragment shaders cannot send or receive information to/from any other neighbouring fragments. Effects where neighbouring pixel values are required can still be achieved, however they may require multiple rendering passes.

Multiple Render Targets

In many cases the output from the graphics pipeline is display onto the screen. However, some graphics systems will actually write to an output framebuffer which may then be accessed in a different rendering pipeline (or rendering pass).

This led to the idea of rendering to ***multiple render targets*** where a single pass through a graphics pipeline can actually render to multiple different targets which can then be used in later passes. These different targets must be the same dimensions and may have to have the same bit depth.

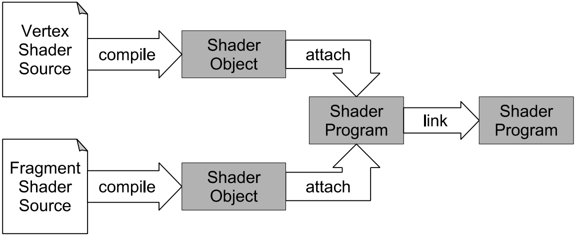
Creating an OpenGL Pipeline

Creating an OpenGL pipeline requires quite a lot of work before we can get anything to display on the screen.

- Load and compile shaders into shader objects.
- Compose and link the shaders into a shader program.
- Create and initialise buffers for our object data.
- Generate OpenGL draw calls using the buffer objects and shader program.

Compiling Shaders

The first step is to load and compile the OpenGL shaders into a shader program. This shader program can then be used to render objects to the screen.



Shaders are usually compiled at run-time but from OpenGL 4.1 these can be loaded from a cached binary file.

Simple Vertex Shader

The following is an extremely simple vertex shader that simply takes a 3D vector as input and outputs it as the position.

```
// OpenGL 3.3
#version 330

// Input to Vertex Shader
in vec3 vert_Position;

void main() {
    //———————————————————————————————————————————————
    // Vertex Position
    //———————————————————————————————————————————————
    gl_Position = vec4(vert_Position, 1.0f);
}
```

This shader does no proper conversion between coordinate systems.

Simple Fragment Shader

This is a very simple fragment shader that takes no input and always sets the pixel colour to white.

```
// OpenGL 3.3
#version 330

// Output from Fragment Shader
out vec4 pixel_Colour;

void main () {
    //----------------------------------------------
    // Fragment Colour
    //----------------------------------------------
    pixel_Colour = vec4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

Compiling Shaders

When compiling shader source code into a shader object you should perform the following steps:

- Create a shader object.
- Attach the shader source code.
- Compile the shader.
- Check the compile status.
- In case of error, print the shader info log.

If you don't test for (and print) a compile error then your program will not work but will not give you feedback on why.

Compiling Shaders

```
// Load and Compile Shader from source file
GLuint loadShader(GLuint type, const char *filename) {
   // Read the shader source from file
   char *source = readFile(filename);

   // Check shader source
   if(source == 0) {
      // Return Error
      return 0;
   }

   // Create the OpenGL Shader
   GLuint shader = glCreateShader(type);

   // Load the source into the shaders
   glShaderSource(shader, 1, &source, NULL);

   // Compile the Shaders
   glCompileShader(shader);
   ...
```

Compiling Shaders

```
// Load and Compile Shader from source file
GLuint loadShader(GLuint type, const char *filename) {
    ...
    // Check shaders for errors
    if(checkShader(shader) == GL_TRUE) {
        // Log
        std::cout << "Loaded: " << filename << std::endl;
    } else {
        // Print Error
        std::cerr << "Error: could not compile " << filename << std::endl;

        // Delete shader source
        delete[] source;

        // Return Error
        return 0;
    }

    // Delete shader source
    delete[] source;

    // Return shader
    return shader;
}
```

Compiling Shaders

```
// Check the compile status of a Shader
GLuint checkShader(GLuint shader) {
   // Check compile status
   GLint status = 0;
   glGetShaderiv(shader, GL_COMPILE_STATUS, &status);

   // Error detected
   if(status != GL_TRUE) {
      // Get error message length
      int size;
      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &size);

      // Get error message
      char *message = new char[size];
      glGetShaderInfoLog(shader, size, &size, message);

      // Print error message
      std::cerr << message << std::endl;

      // Delete message
      delete[] message;

      // Return error
      return GL_FALSE;
   }
   // Return success
   return GL_TRUE;
}
```

Creating a Shader Program

```
// Load Shader Program from shader files
GLuint loadProgram(const char *vert_file, const char *frag_file) {
    // Create new OpenGL program
    GLuint program = glCreateProgram();

    // Shader Handles
    GLuint vert_shader = 0;
    GLuint frag_shader = 0;

    // Load Shaders
    if(vert_file != NULL) vert_shader = loadShader(GL_VERTEX_SHADER,   vert_file);
    if(frag_file != NULL) frag_shader = loadShader(GL_FRAGMENT_SHADER, frag_file);

    // Attach shaders
    if(vert_shader != 0) glAttachShader(program, vert_shader);
    if(frag_shader != 0) glAttachShader(program, frag_shader);

    ...
```

Creating a Shader Program

```cpp
// Load Shader Program from shader files
GLuint loadProgram(const char *vert_file, const char *frag_file) {
   ...
   // Check Vertex Shader
   if(vert_shader == 0) {
      // Print Error
      std::cerr << "Error: program missing vertex shader." << std::endl;

      // Delete Shaders
      if(vert_shader != 0) glDeleteShader(vert_shader);
      if(frag_shader != 0) glDeleteShader(frag_shader);

      // Return Error
      return 0;
   }

   // Check Fragment Shader
   if(frag_shader == 0) {
      // Print Error
      std::cerr << "Error: program missing fragment shader." << std::endl;

      // Delete Shaders
      if(vert_shader != 0) glDeleteShader(vert_shader);
      if(frag_shader != 0) glDeleteShader(frag_shader);

      // Return Error
      return 0;
   }
   ...
```

Creating a Shader Program

```
// Load Shader Program from shader files
GLuint loadProgram(const char *vert_file, const char *frag_file) {
   ...
   // Link program
   glLinkProgram(program);

   // Delete Shaders (no longer needed)
   if(vert_shader != 0) glDeleteShader(vert_shader);
   if(frag_shader != 0) glDeleteShader(frag_shader);

   // Check program for errors
   if(checkProgram(program) == GL_TRUE) {
      // Print Log
      std::cout << "Loaded: program" << std::endl;
   } else {
      // Print Error
      std::cerr << "Error: could not link program" << std::endl;

      // Return Error
      return 0;
   }

   // Return program
   return program;
}
```

Creating a Shader Program

```
// Check the status of a Program
GLuint checkProgram(GLuint program) {
   // Check link status
   GLint status = 0;
   glGetProgramiv(program, GL_LINK_STATUS, &status);

   // Error detected
   if(status != GL_TRUE) {
      // Get error message length
      int size;
      glGetProgramiv(program, GL_INFO_LOG_LENGTH, &size);

      // Get error message
      char *message = new char[size];
      glGetProgramInfoLog(program, size, &size, message);

      // Print error message
      std::cerr << message << std::endl;

      // Delete message
      delete[] message;

      // Return error
      return GL_FALSE;
   }

   // Return success
   return GL_TRUE;
}
```

Object Data

Now that we have defined an OpenGL pipeline, we need some object data to display using it - stored in OpenGL using buffer objects.

Buffer objects are stored in the GPU memory and can be accessed by the shaders of the pipeline. The buffer objects can be bound to different binding points or targets depending on what they are used for.

As a first example, we are going to create a single triangle using three vertexes.

Vertex Buffer Object

There are a number of different targets a buffer object may be bound to. The `GL_ARRAY_BUFFER` is used for buffers of generic data such as the vertex positions in this buffer object.

```
// Triangle Vertexes
GLfloat buffer[9];

buffer[0] =  0.0f; buffer[1] =  0.577f; buffer[2] =  0.0f;
buffer[3] =  0.5f; buffer[4] = -0.289f; buffer[5] =  0.0f;
buffer[6] = -0.5f; buffer[7] = -0.289f; buffer[8] =  0.0f;

// Vertex Array Object (VAO)
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Generate Vertex Buffer Object (VBO)
GLuint vbo = 0;
glGenBuffers(1, &vbo);

// Bind VBO to data target
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Load Vertex Data into VBO
// (target, size, data, usage)
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(GLfloat), buffer, GL_STATIC_DRAW);
```

Vertex Buffer Object

When the data is copied to the buffer, a usage hint is provided that is used to decide where to store the buffer based on:

Frequency of access:

- `STATIC` - written once, used many times.
- `STREAM` - written once, used a few times.
- `DYNAMIC` - written repeatedly, used many times.

Nature of access:

- `DRAW` - written by application, used for drawing by GL.
- `READ` - written by GL, read by application
- `COPY` - written by GL, used for drawing by GL.

Vertex Attributes

We must also provide OpenGL with information on how to access data for each vertex attribute. This must be done for each different attribute.

```
// Get Position Attribute location (must match name in shader)
GLuint posLoc = glGetAttribLocation(program, "vert_Position");

// Set Vertex Attribute Pointer
// (index, size, type, normalised, stride, offset)
glVertexAttribPointer(posLoc, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), NULL);

// Enable Vertex Attribute Array
glEnableVertexAttribArray(posLoc);
```

Vertex Attributes

It is also possible to define the index of the attribute directly in the shader.

```
// Input to Vertex Shader
layout(location = 0) in vec3 vert_Position;
```

Which can then be referenced in the application.

```
// Set Vertex Attribute Pointer
// (index, size, type, normalised, stride, offset)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), NULL);

// Enable Vertex Attribute Array
glEnableVertexAttribArray(0);
```

Drawing Primitives

Once the buffer objects have been appropriate configured, initialised (and bound) we can draw primitives using them. In this case we draw triangles, starting at index 0 and using 3 vertexes.

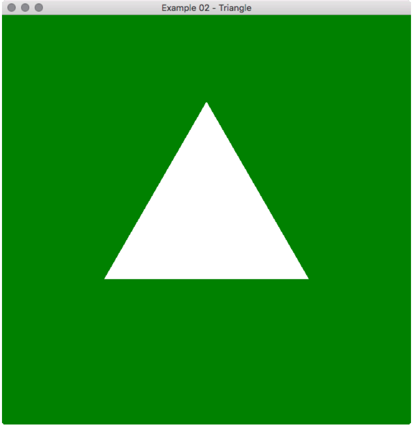
```
// Use the contents of the current buffers to draw triangles
// (mode, first index, number of vertexes
glDrawArrays(GL_TRIANGLES, 0, 3);
```

The drawing mode may be one of:

- `GL_POINTS, GL_LINES, GL_TRIANGLES`
- `GL_LINE_STRIP, GL_TRIANGLE_STRIP`
- ...

Example 02 - Triangle

Put together the output should look something like this:



OpenGL Immediate Mode

If you look up OpenGL code or examples online, you may discover code such as the following:

```
// Start drawing triangles
glBegin(GL_TRIANGLES);

// Draw three vertexes
glVertex3f( 0.0f,  0.577f, 0.0f);
glVertex3f( 0.5f, -0.289f, 0.0f);
glVertex3f(-0.5f, -0.289f, 0.0f);

// Stop drawing triangles
glEnd(GL_TRIANGLES);
```

DO NOT USE THIS METHOD!

OpenGL Immediate Mode

This is called ***Immediate Mode*** as the vertexes of the polygons are defined inside a `glBegin` and `glEnd` block and are drawn as soon as they are defined.

This mode of drawing is now ***obsolete*** and while it may still be supported by some drivers will likely be removed from future versions. While you may still find examples using this drawing method online you should not use any immediate mode drawing in this course.

OpenGL Display Lists

Another ***obsolete*** drawing method you may come across online are OpenGL Display Lists.

```
// Generate a Display List
GLuint list = glGenLists(1);
glNewList(list, GL_COMPILE);

// Start generating list (triangles)
glBegin(GL_TRIANGLES);

// Draw three vertexes
glVertex3f( 0.0f,  0.577f, 0.0f);
glVertex3f( 0.5f, -0.289f, 0.0f);
glVertex3f(-0.5f, -0.289f, 0.0f);

// Finish generating list
glEnd();
glEndList();

// Draw the display list
glCallList(list);
```

DO NOT USE THIS METHOD EITHER!

OpenGL Display Lists

Display Lists are essentially a group of OpenGL commands that have been stored and can be executed again with a single command.

Both this method and Immediate Mode should not be used and you must use ***vertex buffer objects*** for all drawing commands in this course.

Polygon Mode

There are some parts of the non-programmable parts of the OpenGL graphics pipeline that we can still configure. For example, the rasterisation step can be configured using:

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

The options for this are:

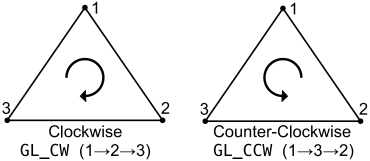
- `face` - `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`
- `mode` - `GL_FILL`, `GL_LINE`, `GL_POINT`

Winding Order

The *front* and *back* of a face is determined the ***winding order*** of the faces. This can be configured with:

```
void glFrontFace(GLenum mode);
```

The winding order can be set to either clockwise or counter-clockwise with the values `GL_CW` and `GL_CCW` respectively.



Back-Face Culling

The winding order can also be used to perform ***back-face culling*** which will not draw faces when the back of the face is seen.

Face culling must be enabled with `glEnable(GL_CULL_FACE);` and the front, back (or even both) can be culled with the following:

```
void glCullFace(GLenum mode);
```

The options for this are:

- `GL_FRONT` - cull front faces.
- `GL_BACK` - cull back faces.
- `GL_FRONT_AND_BACK` - cull all faces.

Multiple Attributes

To achieve more interested graphical effects, we need to provide more vertex attribute data to the shaders.

These attributes may represent a sorts of information about a vertex, for now we will just look at how we associate a colour with each of the vertexes in our triangle.

We start by defining our new shaders.

Vertex Shader

This vertex shader takes two vertex attributes as input - position and colour. It outputs the colour to the fragment shader.

```glsl
// OpenGL 3.3
#version 330

// Input to Vertex Shader
in vec3 vert_Position;
in vec3 vert_Colour;

// Output to Fragment Shader
out vec3 frag_Colour;

void main() {
    // Vertex Colour
    frag_Colour = vert_Colour;

    //----------------------------------------------
    // Vertex Position
    //----------------------------------------------
    gl_Position = vec4(vert_Position, 1.0f);
}
```

Fragment Shader

This fragment shader receives a colour value from the vertex shader (may be interpolated between values) and outputs a colour for the fragment.

```
// OpenGL 3.3
#version 330

// Input from Vertex Shader
in vec3 frag_Colour;

// Output from Fragment Shader
out vec4 pixel_Colour;

void main () {
    //----------------------------------------------
    // Fragment Colour
    //----------------------------------------------
    pixel_Colour = vec4(frag_Colour, 1.0f);
}
```

Multiple Attributes

The vertex data that is loaded into the buffer must be changed to define both the positions and also the colours of the vertexes. In this example we will interleave the positions and colours in the same array.

pos_1	col_1	pos_2	col_2	pos_3	col_3
(x,y,z)	(r,g,b)	(x,y,z)	(r,g,b)	(x,y,z)	(r,g,b)

The positions and colours are all defined with floating point values so can be stored in a single array of floats.

Multiple Attributes

The interleaved buffer data can be defined and copied into the vertex buffer object as follows:

```
// Triangle Vertexes (and colours)
GLfloat buffer[18];

buffer[0]  =  0.0f; buffer[1]  =  0.577f; buffer[2]  =  0.0f;
buffer[3]  =  1.0f; buffer[4]  =  0.0f;   buffer[5]  =  0.0f;
buffer[6]  =  0.5f; buffer[7]  = -0.289f; buffer[8]  =  0.0f;
buffer[9]  =  0.0f; buffer[10] =  1.0f;   buffer[11] =  0.0f;
buffer[12] = -0.5f; buffer[13] = -0.289f; buffer[14] =  0.0f;
buffer[15] =  0.0f; buffer[16] =  0.0f;   buffer[17] =  1.0f;

// Vertex Array Object (VAO)
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Vertex Buffer Object (VBO)
GLuint vbo = 0;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// Load Vertex Data
glBufferData(GL_ARRAY_BUFFER, 18 * sizeof(GLfloat), buffer, GL_STATIC_DRAW);
```

Multiple Attributes

The vertex attributes must then be configured and enabled. Both the position and colour attributes have a stride of 6 and the colour has an offset of 3 (elements).

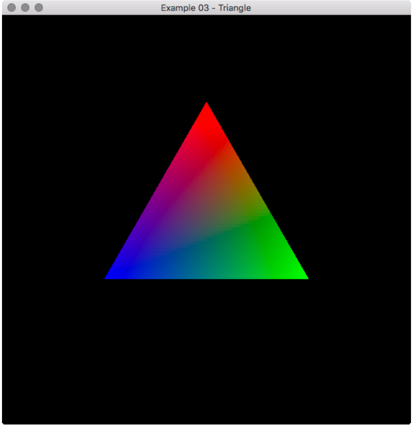
```
// Get Attribute locations (must match names in shader)
GLuint pos = glGetAttribLocation(program, "vert_Position");
GLuint col = glGetAttribLocation(program, "vert_Colour");

// Set Vertex Attribute Pointers (note the stride and offset).
glVertexAttribPointer(pos, 3, GL_FLOAT, GL_FALSE,
                      6 * sizeof(GLfloat), NULL);
glVertexAttribPointer(col, 3, GL_FLOAT, GL_FALSE,
                      6 * sizeof(GLfloat), (GLvoid*)(3*sizeof(GLfloat)));

// Enable Vertex Attribute Arrays
glEnableVertexAttribArray(pos);
glEnableVertexAttribArray(col);
```

Example 03 - Triangle

The example should now display a triangle with the colour attributes passed into the pipeline.



Indexed Drawing

The method we have been using for drawing elements to the screen `glDrawArrays()` draws vertexes in the order they are defined in the buffer. The downside of this approach is that vertexes cannot be reused - a vertex must be defined multiple times to be used for multiple triangles (or other primitives).

The other approach is to define an element buffer that contains a series of indexes in the vertex buffer to define primitives. This will not be more simple when drawing a single triangle but will be very useful when drawing more complex objects.

Indexed Drawing

For our simple triangle example we only need to define three indexes in the element buffer (the target is the `GL_ELEMENT_ARRAY_BUFFER`).

```
// Triangle Indexes
GLuint indexes[3];
indexes[0] = 0;
indexes[1] = 1;
indexes[2] = 2;

// Element Buffer Object (EBO)
GLuint ebo = 0;
glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);

// Load Element Data
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 3*sizeof(GLuint), indexes, GL_STATIC_DRAW);

...

// Draw Elements (Triangles)
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, NULL);
```