GLSL & the Vertex Shader

Game 300

James Dupuis

Objectives

- Learn about:
  - GLSL
  - A shaders code structure
  - How a shader is used
  - What the vertex shader is?
  - How to implement a vertex shader

GLSL Review

- Written in it's own language (GLSL)
  - OpenGL Shading Language
  - C like
  - compiler is built into OpenGL
  - create Shader Objects produces from GLSL code
    - shader objects get linked together to create a Program Object
  - Shaders have different Shader types known as Stages:
    - Vertex shaders (example to follow)
      - required as a minimum
    - Fragment shaders (example to follow)
      - very important, nothing renders without this one.
    - Tesselation shaders
    - Evaluation shaders
    - Geometry shaders
    - Compute shaders
  - Shader stages answer the questions of Who, What, When and Where of drawing to the screen.

breakdown of a simple shader

```
// declare the version the compiler should interpret this as 450 = 4.5
#version 450 core

//no params passed to main function
void main(void)
{
}
```

- #version indicates the minor and major versions used to compile the GLSL code
  - Note GLSL and OpenGL do have different version numbers although they are typically kept in synch.

- Note the main function is the entry point for the code
- Void main(void)
- At a minimum all shaders will contain this same basic construction.

VERTEX SHADER

- Vertex Shader: (WHERE?)
  - used to define the vertex where the following shader elements will be performed.
  - to make a vertex shader, we can use the core code previously shown and add in an assignment to the variable gl_Position:
    - this is a built in variable that OpenGL knows about by default
    - you don't need to declare it
    - assigning to it, assigns a value to the pipeline
    - our vertex shader wants a **single** vertex being produced.
    - by assigning to gl_Position it's the same as if we had a function returning a Vector4

```
#version 450 core
1
2   void main(void)
3   {
4       // gl_Position is a fixed function of Open GL
5       // sets the position for the following steps to do their thing
6       gl_Position = vec4(0.0f, 0.0f, 0.5f, 1.0f);
7   }
```

Vertex Point Shader

- In the Previous Shader example we could use the glDrawArrays(GL_POINTS, 0, 1);
  - Can it be switched to draw a GL_LINES or GL_TRIANGLES?
  - Would we need to change anything?
  - Draw call has 1 as the final param for size / number of vertices?
  - Because our shader only has 1 vert4 specified, each point of the triangle will render at that exact location....
    - sooo our triangle is still a dot
  - To fix this we need to modify the vertex shader to have more than one vertex for the three points of the triangle.

Triangle Vertex Shader

```
void main(void)
{
    // hard-coded traingle points positions
    const vec4 triangleVerts[3] = vec4[3]( vec4 (0.5f,0.5f,0.5f,1.0f),
                                           vec4 (-0.5f,0.5f,0.5f,1.0f),
                                           vec4 (-0.5f,-0.5f,0.5f,1.0f));

    // use the individual triangle verts with the gl_VertexID fixed variable
    // variable is increased each pass through the shader.
    gl_Position = triangleVerts[gl_VertexID];
}
```

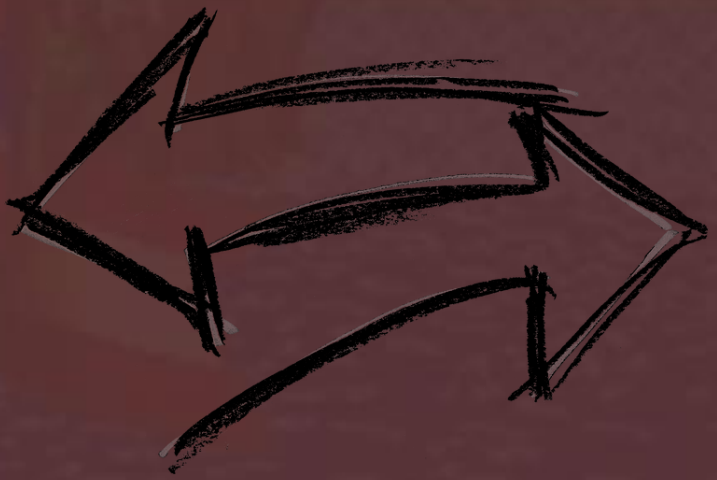
Broken down in next slide

What just happened?

- Shaders have stages, stages communicate and are processed slightly differently.
- For a vertex shader like this, each vertex is a new pass through the shader.
  - The shader just produces a single vertex point on the screen.
- Within our vertex shader is a hidden atomic counter called **gl_VertexID** similar to how **gl_Position** is built into the OpenGL system and available to use.
  - Every time we have a new vertex rendered it increments the counter processing through the shader so we can alter the **gl_Position** to equal the array at index **gl_VertexID**
- The second parameter of the **glDrawArrays**() call indicates how many passes should be made through the shader.
  - So even if we had an array of 4 Vec4 points but we specified 2 in the glDrawArrays call, only 2 points would be drawn.

The in's and out's of shaders...

- So we can create a shader which can draw a triangle, how do we pass values from our application to the shader to have those values change?
- using what's called storage **qualifiers**
  - storage qualifiers are the **in**'s and **out**'s of our shaders... literally.
  - if we want to take input for the shader, define variables as **in**.
  - if you want to output variables from a shader, define variables as **out**.
  - variables with a storage qualifier preceding them are called **Attributes**
- Example:
  - If we modify the vertex shader, we can add in a new "**in**" variable to take in a float for the size of the triangle.
    - in GLfloat size;



Dynamic Values

```
1
2   #version 450 core
3
4   // layout (location=val) defines which param it is passed in from the application
5   // the in dictates it's a parameter passed in.
6   layout (location = 0) in   float size;
7
8   void main(void)
9   {
10      //use the size varialble
11      const vec4 triangleVerts[3] =     vec4[3]( vec4 (size,size, 0.5f,1.0f),
12                                        vec4 (-size,size,0.5f,1.0f),
13                                        vec4 (-size,-size,0.5f,1.0f));
14
15      gl_Position = triangleVerts[gl_VertexID];
16  }
```

*Layout(location = 0) described more later in this lecture

Setting Values

- To supply the float to the vertex fetching stage which will then move it on to the vertex shader we need to tell it what to use in the actual application.
  - To do so, we use:
    - void glVertexAttrib1f(GLuint index, GLfloat v0);
  - each attribute type would require a different call to send each variable.
  - other calls available are:
    - void glVertexAttrib1s(GLuint index, GLshort v0);
    - void glVertexAttrib1d(GLuint index, GLdouble v0);
    - void glVertexAttribI1i(GLuint index, GLint v0);
    - void glVertexAttribI1ui(GLuint index, GLuint v0);
    - void glVertexAttrib2f(GLuint index, GLfloat v0, GLfloat v1);

Program Code (not shader code)

```
1   virtual void render(double currentTime)
2   {
3       GLfloat black[] = {0.0f,0.0f,0.0f,0.0f,1.0f};
4
5       glClearBufferfv(GL_COLOR, 0, colour);
6       glUseProgram(ourShaderProgram);
7
8       //Set the float for the size in the vertex shader
9       glVertexAttrib1f( 0, 0.25f);
10
11      glDrawArrays(GL_POINTS, 0, 1);
12  }
```

Setting Values Cont

- You can even pass a pointer:
  - void glVertexAttrib1fv(GLuint index, const GLfloat *v);
- for vec4 of floats:
  - void glVertexAttrib4fv(GLuint index, const GLfloat *v);
- full list here:
  - https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttrib.xhtml

- Q: what's the first parameter for?
- A: this tells the vertex fetching stage what parameter it is
  - you need to indicate the order of variables inside your shader as well...
  - it doesn't just assume the declaration order is the order.
  - to do this we use a **layout qualifier** (seen previously in the triangle size example)

Layout Qualifiers

- Simply add the following **before** your in variable declaration:
  - layout (location = 0)
- Example:
  - layout (location = 0) in float size;

- Now our shader knows the Attribute set to location 0 should be assigned to this float.
- We assign that in our program using:
  - glVertexAttrib1f(0, someFloatSize);

- *Note: This should be done after the glUseProgram call, but before the glDraw call.

Dynamic Values

```
1
2   #version 450 core
3
4   // layout (location=val) defines which param it is passed in from the application
5   // the in dictates it's a parameter passed in.
6   layout (location = 0) in   float size;
7
8   void main(void)
9   {
10      //use the size varialble
11      const vec4 triangleVerts[3] =     vec4[3]( vec4 (size,size, 0.5f,1.0f),
12                                        vec4 (-size,size,0.5f,1.0f),
13                                        vec4 (-size,-size,0.5f,1.0f));
14
15      gl_Position = triangleVerts[gl_VertexID];
16  }
```

Setting Values Cont

- Now we can supply a variable to the vertex shader,
- This only works for communicating information to the vertex shader.
  - you can't directly set values from the program to individual stages/shaders using the in attributes.
    - We'll cover uniforms later which handles this.
  - Data can, however, flow through the rendering pipeline.
  - If you want to send a value to the next shader in the pipeline, it first can be sent to the vertex shader through an **in** declared variable and forwarded along
  - especially useful if the results of the vertex shader or other shaders along the line impact the data required to be passed along.

Out variables

- You can also use the same **in** & **out** qualifiers to pass data from stage to stage
  - as an example you may want the vertex shader to forward a color attribute to the fragment shader.
  - to do this, use the **out** variable on the first shader (vertex) and have an **in** variable defined on the subsequent shader (fragment in this case).
  - there is **no** layout (location) requirements for these inter-stage data transfers.
  - only requirement is that the variables be named the same on both the **in** and **out** and have the same type.

Program Code

```
1  virtual void render( double currentTime )
2  {
3      GLfloat black[] = { 0.0f, 0.0f, 0.0f, 0.0f, 1.0f };
4      glClearBufferfv( GL_COLOR, 0, colour );
5      glUseProgram( ourShaderProgram );
6
7      // Set the float for the size in the vertex shader
8      glVertexAttrib1f( 0, 0.25f );
9
10     // Set the colour of the triangle
11     GLfloat colour[] = { 1.0f, 0.0f, 0.0f, 1.0f };
12     glVertexAttrib4fv( 1, colour );
13
14     glDrawArrays( GL_POINTS, 0, 1);
15 }
```

Vertex Shader Code

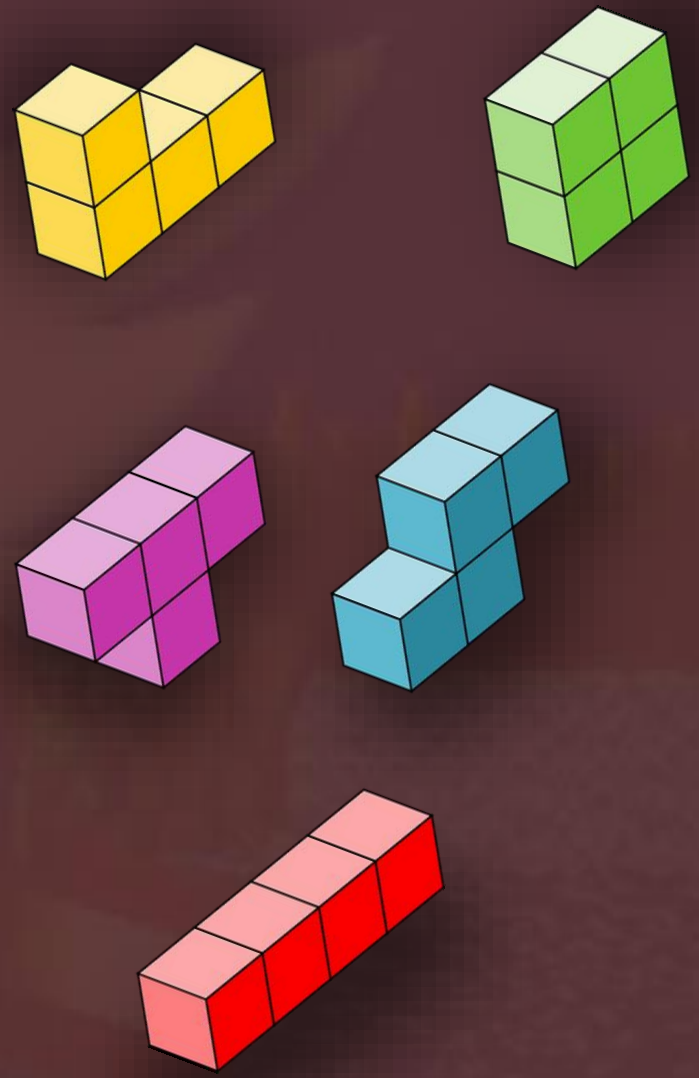
```
1  #version 450 core
2
3  layout ( location = 0 ) in   float size;
4
5  // grab the size from the application
6  layout ( location = 1 ) in vec4 colour;
7
8  // this will be forwarded onto the fragment shader
9  out vec4 fwd_colour;
10
11 void main(void)
12 {
13     //remember to assign to the variable being forwarded
14     fwd_colour = colour;
15
16     const vec4 triangleVerts[3] = vec4[3] ( vec4 (size, size, 0.5f, 1.0f ),
17                                             vec4 ( -size, size, 0.5f, 1.0f ),
18                                             vec4 ( -size, -size, 0.5f, 1.0f ) );
19
20     gl_Position = triangleVerts[ gl_VertexID ];
21 }
```

fragment Shader Code

```
1  #version 450 core
2  // grab the colour passed from the vertex shader
3  in vec4 fwd_colour;
4
5  // declare the out variable which is used to draw in the next stage
6  out vec4 colour;
7
8  void main(void)
9  {
10     // copy forwarded colour which came from the application through the vertex shader.
11     colour = fwd_colour;
12 }
```

Interface Blocks

- **Interface Blocks** are the structs of shaders
  - used to group a set of variables which are passed from shader stage to shader stage
  - this can include structures themselves.
  - structure of Interface Block must match on in and out declarations
    - variable name may be altered.
  - Cannot be used as an **in** for **vertex shaders** or **outs** of **fragment** shaders (the two programmable ends of the pipeline)



Interface Blocks (Shader #1)

```
1  #version 450 core
2
3  //interface block forwarded to the fragment shader
4  out INTERFACE_BLOCK_OUT
5  {
6      vec4 colour;
7  } int_blck;
8
9  void main(void)
10 {
11     //example of assigning a value to the variable inside interface block
12     int_blck.colour = vec4( 1.0f, 0.0f, 0.0f, 1.0f);
13 ...
14 }
```

Interface Blocks (Shader #2)

```
1   #version 450 core
2
3   //interface block received from the vertex shader
4   in INTERFACE_BLOCK_OUT
5   {
6       vec4 colour;
7   } int_blck;
8
9   void main(void)
10  {
11      //example of using value contained inside interface block
12      color = int_blck.colour;
13  ...
14  }
```

Summary

- Shaders are the main point of processing data and creating everything you see on screen.
  - Understanding how to create a basic shader and how to pass data to them is an important requirement.
  - The vertex shader is the main point of contact to process the vertices of your models and worlds.
    - Which is a minimal requirement for all shader program objects.

- Today we Learned about:
  - Shaders
  - GLSL code format
  - In declared variables
  - Layout qualifiers
  - out variables
  - Interface Blocks