First OpenGL Shader Program

GAME 300

James Dupuis

Objectives

- Learn about:
  - Process the steps to create an OpenGL Program Object
  - Process the steps to create / load GLSL Shaders Objects
  - VAO

OpenGL Program

- Program Objects
  - An OpenGL object
  - Contains compiled shader code
  - Manages Draw calls in Context
  - Needs to be cleaned up in the shutdown code / applications destructor

- Declared as a GLuint
  - GLuint program_object;

```
35
36      virtual void startup()
37      {
38          program = glCreateProgram();
39      }
```

- Created using glCreateProgram();
  - Program_object = glCreateProgram();
  - This creates an empty program which is pretty useless for now.
  - We need to supply our program shaders for it to be able to do something.
  - This should be done inside the **startup**() function not Init()
    - Init happens before the context is initialized and Startup happens after.

Shaders

- Shaders are a component of the rendering Pipeline.
  - Follow GLSL
    - Unique language although very similar to C/ C++
    - Not able to perform recursion
    - Can be directly inline code as a constant string
      - Clutters up the code when shaders become complicated
    - Often housed in external files
    - Use the extension of .glsl
      - Some developers use an abbreviated extension for each type of shader for clarity.
        - .vert - a vertex shader
        - .tesc - a tessellation control shader
        - .tese - a tessellation evaluation shader
        - .geom - a geometry shader
        - .frag - a fragment shader
        - .comp - a compute shader
- Example: https://www.shadertoy.com/view/4slSWf

Shader use

- The right is an example of inline Shader code.
- Shaders have a 4 step process to be used:
  - Load/Initialize Code
  - Compile Code
  - Attach compiled Shader object to Program Object
  - Delete Shader Object
- The right shows an example inline code of 2 shader files.
  - This is to illustrate that shaders are just txt files to our cpp compiler.

```
36  virtual void startup()
37  {
38      static const char * vs_source[] =
39      {
40          "#version 450 core                                                  \n"
41          "                                                                   \n"
42          "void main(void)                                                    \n"
43          "{                                                                  \n"
44          "    const vec4 vertices[] = vec4[](vec4( 0.25, -0.25, 0.5, 1.0),  \n"
45          "                                   vec4(-0.25, -0.25, 0.5, 1.0),  \n"
46          "                                   vec4( 0.25,  0.25, 0.5, 1.0)); \n"
47          "                                                                   \n"
48          "    gl_Position = vertices[gl_VertexID];                           \n"
49          "}                                                                  \n"
50      };
51
52      static const char * fs_source[] =
53      {
54          "#version 450 core                                                  \n"
55          "                                                                   \n"
56          "out vec4 color;                                                    \n"
57          "                                                                   \n"
58          "void main(void)                                                    \n"
59          "{                                                                  \n"
60          "    color = vec4(0.0, 0.8, 1.0, 1.0);                              \n"
61          "}                                                                  \n"
62      };
63
64      program = glCreateProgram();
65  }
66
```

Initializing / Loading Shaders

- OpenGL Applications often include a shader loader
  - Loads external shader code into a char*.
  - Const char* shaderSrcCode = LoadShader(myshaderfile.vert);
- Shaders Objects are stored into the same type of variable as an OpenGL Program Object:
  - GLuint vertexShader;
  - This will be used to store the compiled shader.
  - Use the function glCreateShader() to create the empty shader object similar to glCreateProgram()

Shader Enum

- Example:

glCreateShader(GL_VERTEX_SHADER);

- Parameter:
  - Single GLEnum value which dictates which type of shader is going to be applied.
- Types of Shader options:
  - GL_COMPUTE_SHADER,
  - GL_VERTEX_SHADER,
  - GL_TESS_CONTROL_SHADER,
  - GL_TESS_EVALUATION_SHADER,
  - GL_GEOMETRY_SHADER,
  - GL_FRAGMENT_SHADER.
- Returns:
  - An index (handle) to the shader pointer tracked by OpenGL as a GLuint

```
virtual void startup()
{
    static const char * vs_source[] =
    {
        "#version 450 core                                                  \n"
        "                                                                   \n"
        "void main(void)                                                    \n"
        "{                                                                  \n"
        "    const vec4 vertices[] = vec4[](vec4( 0.25, -0.25, 0.5, 1.0),   \n"
        "                                   vec4(-0.25, -0.25, 0.5, 1.0),   \n"
        "                                   vec4( 0.25,  0.25, 0.5, 1.0)); \n"
        "                                                                   \n"
        "    gl_Position = vertices[gl_VertexID];                           \n"
        "}                                                                  \n"
    };

    static const char * fs_source[] =
    {
        "#version 450 core                                                  \n"
        "                                                                   \n"
        "out vec4 color;                                                    \n"
        "                                                                   \n"
        "void main(void)                                                    \n"
        "{                                                                  \n"
        "    color = vec4(0.0, 0.8, 1.0, 1.0);                              \n"
        "}                                                                  \n"
    };

    GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
    GLuint vs = glCreateShader(GL_VERTEX_SHADER);


    program = glCreateProgram();
}
```

Compiling Shaders

- Shaders are compiled when the Application is initialized
  - startup function ideally only run once
    - Application Constructor or Init()
  - before any attempt at using the shader

- Two step phase
  1. Associate the shader source char* with the shader object.
     - This is done using the glShaderSource() function
     - Four Parameters:
       1. Shader Object (GLuint)
       2. size, number of shaders being loaded (GLsizei)
       3. Vertex Source Code (GLChar*)
       4. Length of (GLint)

     - Example:
       - glShaderSource (vertexShader, 1, vertexSrcCode, NULL);

  2. Compile the Shader
     1. This is done using the function glCompileShader()
        - This function has 1 parameter, the shader object (GLuint)

```
57              "                                                   \n"
58              "void main(void)                                    \n"
59              "{                                                  \n"
60              "    color = vec4(0.0, 0.8, 1.0, 1.0);              \n"
61              "}                                                  \n"
62          };
63
64          GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
65          GLuint vs = glCreateShader(GL_VERTEX_SHADER);
66
67          glShaderSource(fs, 1, fs_source, NULL);
68          glShaderSource(vs, 1, vs_source, NULL);
69
70          glCompileShader(vs);
71          glCompileShader(fs);
72
73
74          program = glCreateProgram();
75      }
76
```

Attach Shaders

- Shaders objects need to be attached to a Program Object to be useful.
  - This is done with the function glAttachShader()
    - Takes two Parameters:
      1. The program Object – created this earlier, the basic GLuint assigned to using glCreateProgram();
      2. The shader you wish to attach to the program object.
      - This is similar to a destination and source.
  - Example:

    glAttachShader(programObject, vertexShader);
    - This should be done inside the Startup() function.

```
59              "{                                                          \n"
60              "    color = vec4(0.0, 0.8, 1.0, 1.0);                      \n"
61              "}                                                          \n"
62          };
63
64          GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
65          GLuint vs = glCreateShader(GL_VERTEX_SHADER);
66
67          glShaderSource(fs, 1, fs_source, NULL);
68          glShaderSource(vs, 1, vs_source, NULL);
69
70          glCompileShader(vs);
71          glCompileShader(fs);
72
73          program = glCreateProgram();
74
75          glAttachShader(program, vs);
76          glAttachShader(program, fs);
77
78      }
```

Delete Shaders

- Shader Objects need to be deleted after being attached to the OpenGL Program.
  - They are no longer needed after being attached
  - Using memory space.
- To delete a shader use the glDeleteShader() function.
  - Requires 1 parameter: the shader object to delete.
  - Example:
    - glDeleteShader(vertexShader);
    - This should be done inside the **Startup**() function not the **Shutdown()** unless the shader is kept as a member variable to the application
- Source GLSL shader code can be freed normally assigning NULL to the char*.
  - This can be done as soon as the shader has been compiled into the shader object

```
61              "}                                                    \n"
62          };
63
64          GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
65          GLuint vs = glCreateShader(GL_VERTEX_SHADER);
66
67          glShaderSource(fs, 1, fs_source, NULL);
68          glShaderSource(vs, 1, vs_source, NULL);
69
70          glCompileShader(vs);
71          glCompileShader(fs);
72
73          program = glCreateProgram();
74
75          glAttachShader(program, vs);
76          glAttachShader(program, fs);
77
78          glDeleteShader(vs);
79          glDeleteShader(fs);
80      }
```

Linking the Program Object

- With a Shader Object attached to a Program object we have a few steps left:
  - Link the programs shaders using glLinkProgram( )
    - This is the process where the program takes each shader code object and creates an executable
    - Each shader runs on a different processor of the GPU.
    - Example: glLinkProgram(program_object);
    - This should be done inside the **Startup**() function

```
60                color = vec4(0.0, 0.8, 1.0, 1.0);                    \n"
61            "}                                                       \n"
62        };
63
64        GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
65        GLuint vs = glCreateShader(GL_VERTEX_SHADER);
66
67        glShaderSource(fs, 1, fs_source, NULL);
68        glShaderSource(vs, 1, vs_source, NULL);
69
70        glCompileShader(vs);
71        glCompileShader(fs);
72
73        program = glCreateProgram();
74
75        glAttachShader(program, vs);
76        glAttachShader(program, fs);
77
78        glDeleteShader(vs);
79        glDeleteShader(fs);
80
81        glLinkProgram(program);
82    }
```

Completing the Program Object

- An OpenGL application can have multiple shader program objects used.
- We need to tell OpenGL which program is going to be used to do any following manipulations using glUseProgram( )
  - Sets the draw state
  - Example: glUseProgram(program_object);
  - This should be done inside the **Render**() function

```
84      virtual void render(double currentTime)
85      {
86          //static const GLfloat red[] = { 1.0f, 0.0f, 0.0f, 1.0f };
87          GLfloat color[] = { 1.0f, 0.0f, 0.0f, 1.0f };
88
89          glClearBufferfv(GL_COLOR, 0, color);
90
91          glUseProgram(program);
```

Review Program

- Within an Application
  - We have a context
    - Which houses a Program Object
      - Which can house many shaders
      - Compiles shader code at runtime
      - Links shader code to form the Program Object
- Our main application communicates with the Program Object to:
  - Render objects
  - Manipulate what is displayed on screen.

Drawing

- Once we have a program object established and have told OpenGL we are ready to use it (glUseProgram()) we can make calls directly using drawing functions:
- glDraw...
  - There are a bunch of glDraw functions available
  - start with simplicity in glDrawArrays.
  - glDrawArrays (GLenum mode, GLint first, GLsizei count)
    - GLenum = what primitive type to draw (GL_POINTS, GL_TRIANGLES, GL_LINES...)
    - GLint = first index in the array.
    - GLsizei = number of vertices to draw/render.

Using OpenGL

```
84      virtual void render(double currentTime)
85      {
86          //static const GLfloat red[] = { 1.0f, 0.0f, 0.0f, 1.0f };
87          GLfloat color[] = { 1.0f, 0.0f, 0.0f, 1.0f };
88
89          glClearBufferfv(GL_COLOR, 0, color);
90
91          glUseProgram(program);
92          glDrawArrays(GL_TRIANGLES, 0, 3);
93      }
94
```

VAO

- Vertex Array Objects supplies input to the vertex shader.
  - required to draw to the screen.
  - Even if your shader has no input, this is a requirement for the shader to run
  - Created with the following function:
    - void glCreateVertexArrays(GLsizei size, GLuint *vertexArray);
      - Introduced in 4.5
      - Previously (glGenVertexArrays())
    - needs to be bound to the current object so Open GL understands where it's being used.
      - (it's context)
    - should keep a class specific reference to the created Vertex Array Objects and a reference to the compiled shaders so that they are only instantiated as need be.

VAO

- Vertex Array Objects are a requirement for a simple Program Object to function
- Declared yet again as:
  - GLuint vao;
- The Vertex Array Objects have 2 main requirements for initial setup:
  - glCreateVertexArrays() & glBindVertexArray()
    - Example: glGenVertexArrays(1, &vao);
    - glBindVertexArray(vao);

VAO

```
71          glCompileShader(fs);
72
73          program = glCreateProgram();
74
75          glAttachShader(program, vs);
76          glAttachShader(program, fs);
77
78          glDeleteShader(vs);
79          glDeleteShader(fs);
80
81          glLinkProgram(program);
82
83          glCreateVertexArrays(1, &vao);
84          glBindVertexArray(vao);
85      }
```

Clean up

- Although not entirely a requirement an Important part of your application to consider is the shutdown() function which will be called before the application closes.
  - We need to call OpenGL specific functions to let it know it's ok to let go…
  - Clean up the Program Object using:
    - glDeleteProgram(program);
  - Clean up the VAO using:
    - glDeleteVertexArrays(1, &vao);
- These objects are not using typical memory of the system but rather GPU memory.

```
 98      virtual void shutdown()
 99      {
100          glDeleteProgram(program);
101          glDeleteVertexArrays(1, &vao);
102      }
103
```

Clean up

- We should also be cleaning up our CPU side with the following:

```
// Cleans up the Context
SDL_GL_DeleteContext( context );


//Cleans up the Windows Allocation
SDL_DestroyWindow( window );
```

Summary

- We've individually pieced together an OpenGL object so that we can render a simple triangle to the screen.
- In doing so we've introduced some major concepts:
  - Application Context
  - Program Objects
  - Shader Objects
  - How they get compiled
  - Vertex Array Objects
  - How to clean up after yourself
- Next Time:
  - Diving into Shaders