Shaders

What is a shader?

A shader is a program that runs on the GPU. In other words, it's something that can be written, linked, compiled, and run like any other program, but OpenGL sends it to the GPU. In graphics programming, it's important to be able to use the power of the GPU to draw graphics to the screen. Not everything should be done on the GPU; sometimes doing calculations on the CPU and sending it to the GPU is better. However, there will be things the GPU will be way faster at, and therefore sending them to the GPU makes more sense.

What types of shaders are there?

Vertex shaders

- Gets called for each vertex
  - In a triangle, will get called 3 times
- Primary purpose is to tell OpenGL where the vertex is
- Also used to pass data (attributes) into the next stage
  - Vertex shader will take in all of the attributes in the buffer
    - First argument of `glVertexAttribPointer();` corresponds to an index in the vertex shader
    - Through that format, the vertex attribute can be accessed
    - Then OpenGL can be told where to position the vertex ### Fragment (Pixel) shader
- Gets called for each pixel that needs to get rasterized
  - Rasterized — When something gets drawn to the screen
- Primary purpose is to tell OpenGL what the color of the pixel is
  - Sort of like a coloring book, shading in the colors of the shape
- Since it gets called so much, even something as simple as 5×5 can slow a program down
  - Doing math in the fragment shader should be minimized in optimization
  - Calculating things like lighting should, however, be done in the fragment shader
  - Essentially, only things that should be done for **every** pixel should be done in the frag shader ### Other There are other types of shaders, but vertex and fragment shaders are the two most common. With just the vert and frag shaders, 90% of the graphics can be done. In fact, most complex game engines generate shaders live at runtime!

  Note: Shaders work the same as the rest of OpenGL: As a state machine. When a shader needs to be used, it has to be enabled, and any data need to be sent to the data in a uniform (which comes from the CPU). More on that later.

Making a shader!

Compiling

The first step in getting a shader working is to compile the source code, so let's write a function to do that (writing a function is recommended to streamline the code). The function to compile the shader should take in the type of shader and its source code, and return an `unsigned int` reference to the shader's ID. The function header should look something like this:

```
static unsigned int CompileShader(unsigned int type, const std::string& source)
```

The type *can* be a `GLenum`, but using that limits the ability to transfer and work between different graphics APIs, so we'll use an `unsigned int`.

Next, we need to create a shader and assign it an ID. This works in much the same way as buffers, but instead of using `glGenBuffers()`, we'll be using the `glCreateShader()` function. This function takes in one argument, which is the type of shader to create. In our case, this is dependent on the `unsigned int type` argument passed to the `CompileShader()` function. Declaration of the variable will look something like the following:

```
unsigned int id = glCreateShader(type);
```

After creating the shader and assigning it an ID, we now need to take in the source code of the shader as a `const char*` and assign the newly created shader its respective source code. Since we're passing in an `std::string&` as our argument, the `std::string.c_str()` function will come in handy. Next, we need to use the `glShaderSource()` function to actually assign the source code to the shader. This function can take in arrays of strings and lengths to better determine how to parse the shader code, but we won't be using that functionality since we'll only be using a single string as source code and we want to use the whole string.

```
const char* src = source.c_str();
glShaderSource(id, 1, &src, nullptr);
```

Finally, we need to call the `glCompileShader()` function to compile the shader, passing in our ID as the shader to parse. Then we're done! Well, almost. Right now, if there's an error in the shader code, OpenGL might just display a black screen or crash, which gives us no real information we can use do diagnose the error. We'll remedy this later, but for now the final `CompileShader()` function should look like this:

```
/* Function to compile shader code into a OpenGL shader */
static unsigned int CompileShader(unsigned int type, const std::string& source)
{
    unsigned int id = glCreateShader(type); // Create a shader with an ID of "id"
    const char* src = source.c_str(); // Convert the source code into a const char*
    glShaderSource(id, 1, &src, nullptr); // Assign the source code to the shader "id"
    // TODO: Error handling

    return id; // Return the compiled shader
}
```

Creating

To create the shader, we'll use our new `CompileShader()` function in another function we'll call `CreateShader()`. This function will have the job of making the actual shader program, which includes both the vertex and fragment shaders. After attaching both shaders to the program, the program will then be linked (just like a C++ program), and also validated to check for any invalid shaders. To do this, we'll make use of the `glLinkProgram()` and `glValidateProgram()` functions. After finishing working with the shaders themselves, we can delete them to free up space using the `glDeleteShader()` function. The arguments to our function will be the shader code for the vert and frag shaders. The function should look something like this:

```
/* Function to create a shader program */
static unsigned int CreateShader(const std::string& vertexShader, const std::string& fragmen
{
    unsigned int program = glCreateProgram(); // Create the program to store both shaders

    unsigned int vs = CreateShader(GL_VERTEX_SHADER, vertexShader); // Create the vertex sha
    unsigned int fs = CreateShader(GL_FRAGMENT_SHADER, fragmentShader); // Create the fragme

    /* Attach the shaders to the program */
    glAttachShader(program, vs);
    glAttachShader(program, fs);

    /* Link and validate the program */
    glLinkProgram(program);
    glValidateProgram(program);

    /* We have no use for the individual shaders now, delete them */
    glDeleteShader(vs);
    glDeleteShader(fs);

    return program; // Return the ID of the finalized program
}
```

Notice the above function replicates the steps needed to create a shader for both a vert and frag shader, not just one. While this may seem obvious, it's important not to omit either from any of the steps involved, otherwise the program might just display a black screen, which is hard to debug. We will, however, be fixing that in the next step.

Errors and Debugging

We can use many functions to check the status of our programs, but one major function used frequently in debugging shaders is `glShaderiv()`. This function queries the shader in any number of ways specified by the user, and returns a result of that query into a user-defined variable. For example, the following code gets the compile status of the shader with the ID `id`, which is either `GL_TRUE` or `GL_FALSE`.

```
int result;
glGetShaderiv(id, GL_COMPILE_STATUS, &result);

if (result == GL_FALSE)
{
    // Get and report the error here
}
```

Once we know there's been a compiler error, actually getting the error and printing it is more complicated then usual. First, we need to get the length of the log string, which can be done again through querying the shader with the `glGetShaderiv()` function. This is very similar to getting the compile status, but instead we are getting the length of the info log so we can display the whole thing later.

```
int length;
glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
```

Next, we need to get the actual message to print to the console. This means first allocating enough memory on the stack to be able to display the message using the length variable we just declared, and then inputting the shader info log data into that memory. Because C++ doesn't want to use a non-constant value to allocate memory on the stack for `char` arrays, we'll use a C function `alloca()` to get the result we want. Then we can just use the `glGetShaderInfoLog()` function to, well, get the shader info log!

```
char* message = (char*)alloca(length * sizeof(char));
glGetShaderInfoLog(id, length, &length, message);
```

Now, having gotten the full message, we can simply print it out to the console, with a little bit of precursory information about which shader is causing the error. The final code should look like this:

```
int result;
glGetShaderiv(id, GL_COMPILE_STATUS, &result);

if (result == GL_FALSE)
{
    int length;
    glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
    char* message = (char*)alloca(length * sizeof(char));
    glGetShaderInfoLog(id, length, &length, message);

    std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "vertex" : "fragment")
    std::cout << message << std::endl;

    glDeleteShader(id);
    return 0;
}
```

This section of code can replace the `// TODO: Error handling` from the **Compiling** step. Although there's now an error message printing to the console, later on we'll use assertions to make no errors in shaders a necessary step. For now, simply printing the error will have to do.

Writing the Shader Code

Shaders in OpenGL are written in GLSL, which stands for GL Shader Language (or something like that). In the next few sections, we'll be going over writing the actual shader code, as well as how we're going to get that shader code from other files into our main file.

GLSL uses syntax that is very alike to C/C++, so it shouldn't be too hard to grasp. Here we'll go line by line through a typical vertex shader, starting from the top. Then we'll briefly run though a similar process for the fragment shader.

Vertex Shader

```
#version 330 core
```

This line basically says to use GLSL version 330, and core means that all functions must not be deprecated. Versions `450` or `440` can be used, but decrease backwards compatibility and come with a lot of new features that we don't necessarily need yet.

```
#version 330 core

void main()
{
}
```

This looks a lot like a standard C++ main function… because it is. This main function will get called when this vertex shader gets called. Now, we'll delve into what's inside this main function.

```
#version 330 core

layout(location = 0) in vec4 position;
```

```
void main()
{
}
```

This new line defines a layout and a new variable to use in our shader. The `layout(location = 0)` bit references our `glVertexAttribPointer()` call earlier, where the first argument defines that attribute's "position". Since we defined it as 0 there, we use `layout(location = 0)` here. The `in vec4 position` basically says that we are taking in a new value from the location we just defined, it's a 4-component vector, and it's called position.

```
#version 330 core

layout(location = 0) in vec4 position;

void main()
{
    gl_Position = position;
}
```

This new line is the line that actually sets the position of the vertex to the correct location on screen. This is done by setting the pre-made attribute of the vertex `gl_Position` to the `position` variable we just passed in. Even though our position is 2-dimensional, OpenGL can interpret that and convert it to a `vec4` quantity for us.

Now we're done writing the code for the vertex shader! Yes, it's that simple. At least, that simple for now. These shaders are really powerful, and you can do some really cool stuff with them.

Fragment Shader

Moving on to the fragment shader, a lot of the code is the same, if not quite similar. We still have the version declaration and the `main()` function, but there are a few changes we need to make. First, we need to be *outputting* a *color*, not *inputting* a *position*. We can do this by changing line 3 of our vertex shader to read:

```
layout(location = 0) out vec4 color;
```

Now, in the main function, we can set the color output variable for each pixel. In this case, just as a test, we'll set every pixel on the shape to be red.

```
color = vec4(1.0, 0.0, 0.0, 1.0);
```

The final fragment shader should have the following code:

```
#version 330 core
```

```
layout(location = 0) out vec4 color;

void main()
{
    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

In the next section, we'll cover how to get this code from files (such as `.shader` files) into the actual C++ program, as well as how to parse, compile, and utilize the code within the files.

Parsing Shaders from Files

Say we create a folder called "res" in our project directory, and we want to store our shader files in a subdirectory called "shaders." We can create these directories and also a file in the shaders directory called "Basic.shader," but natively OpenGL cannot recognize this and use it automatically. Instead, we'll have to manually parse this file (which will contain both vert and frag shader code) so we can use it in our main C++ file.

In order to make our shader file easier to parse, we'll append `#shader <shader type>` at the top of each of our shaders, making our final shader file look like this:

```
#shader vertex
#version 330 core

layout(location = 0) in vec4 position;

void main()
{
    gl_Position = position;
}

#shader fragment
#version 330 core

layout(location = 0) out vec4 color;

void main()
{
    color = vec4(0.1, 0.0, 0.0, 0.1);
}
```

Now that we have our shader in its own contained file, we can port that file into C++ and parse it. To do this, we'll write a function so we can call it neatly later. Before we start writing the function, we'll need to `#include <fstream>` for getting the files into our main program, and `#include <string>` and `include`

`<sstream>` for parsing the actual file. Before we write the function though, let's create a data type to hold the output, which will be two strings (the source code of the vert and frag shader). This can be a simple struct with two variables; it doesn't have to be complex.

```
struct ShaderProgramSource
{
    std::string VertexSource;
    std::string FragmentSource;
}
```

Now that we have our return type set up, we can start to build our function. First, we'll start a file stream for a file defined by an argument passed into the function as follows:

```
static ShaderProgramSource ParseShader(const std::string& filepath)
{
    std::ifstream stream(filepath);
}
```

Next, we'll define an `enum` to hold the mode of our function, which can vary between `NONE`, `VERTEX`, and `FRAGMENT`. This will be used to determine which stream the lines from the file should be put into.

```
enum class ShaderType
{
    NONE = -1, VERTEX = 0, FRAGMENT = 1
};
```

Now we'll start actually going line by line through the file and analyzing its contents. To do this, we'll be using that `sstream` header file we imported earlier. First, let's define a `line` variable to hold the current line we're analyzing. Next, we'll create a string stream (sstream) which will hold the source code of our two shaders temporarily. Then, we'll set the ShaderType to `NONE` by default, and we'll be ready to start parsing!

```
std::string line;
std::stringstream ss[2];
ShaderType type = ShaderType::NONE;
```

In a while loop, we'll go through each line until there are no more lines, and read their contents. If the line contains the `#shader` keyword, we need to then look for the type of shader it's defining and switch the mode to that shader so we know which stringstream we need to put the following code into. If it doesn't, we can just put that line into the current stringstream, followed by a newline character. This is done by the following code:

```
/* Iterate over every line in the file, storing it in the variable "line" */
while (getline(stream, line))
{
    /* If the line contains the string "#shader" at some point */
    if (line.find("#shader") != std::string::npos)
    {
        /* Try to find either "vertex" or "fragment" and set the mode appropriately */
        if (line.find("vertex") != std::string::npos)
            type = ShaderType::VERTEX;
        else if (line.find("fragment") != std::string::npos)
            type = ShaderType::FRAGMENT;
    }
    /* Otherwise, just add the current line to the string stream corresponding to the curren
    else
    {
        ss[(int)type] << line << '\n';
    }
}
```

Finally, after parsing the file and putting the result into the string stream, we can return a `ShaderProgramSource` object, ending our function.

```
return { ss[0].str(), ss[1].str() };
```

Using the Parsed Code in OpenGL

When we wrote the shaders, we didn't put them anywhere in any file, so let's change that! Make a new file in a folder called `res/shaders/`, and call it whatever you want. For now, we'll use `Basic.shader`. Now, copy the shader code from before and put it into the `res/shaders/Basic.shader` file. Once you add the `#shader` headers, the final file should look like this:

```
#shader vertex
#version 330 core

layout(location = 0) in vec4 position;

void main()
{
    gl_Position = position;
}

#shader fragment
#version 330 core

layout(location = 0) out vec4 color;

void main()
{
    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

We can now use the ParseShader function to get the source code from this file in our `main()` function. If we take this code and store it in a `ShaderProgramSource` object, we can pass that into our `CreateShader()` function and actually create the shader:

```
/* Get the source from a file (path is relative to the executable) */
ShaderProgramSource source = ParseShader("res/shaders/Basic.shader");
/* Assign the shader source to a new shader with ID "shader" */
unsigned int shader = CreateShader(source.VertexSource, source.FragmentSource);

/* Use the program when drawing */
glUseProgram(shader);
```

Now, after we finish looping in the window, we need to delete the shader program to free up memory. This can be done as shown below:

```
glDeleteProgram(shader);
```

And we're done! When you run the application, whatever shape you're drawing should be completely red. Now you can write shaders in different files and use them in your main application!

Credits

Where I Learned

- The Cherno
- Learn OpenGL ### Cool Stuff
- Geeks3D Shader Library
- ShaderToy
- GLSL Sandbox
- This list by 'radixzz'