

The Cherno: OpenGL

Engineer:	John Trites January 7 th , 2021
Topic:	The Cherno – OpenGL Tutorials (Daniel Weaver)
Purpose:	Interview Study Guide

Visual Studio 2019 Community: Setup for C++

- Install VS2019 Community IDE
 - Update Ver. 16.8.3
 - Current Ver. 16.2.0
 - Total Space Req'd: 4.79GB → **Update**
 - Restarted PC
- VS2019 Sign-In – new password (1-7-21)
 - jtrites@tritesengserv.com / code = 9187354
 - Pswd: TESI2021
- Create a New Project: Empty Project **C++** **Windows** **Console**
 - Project Name: NewWinProject
 - Location: C:\Dev\NewProject
 - Solution Name: NewWinProject
 - ■ Place solution and project in the directory → **Next**
 - RT+CLK > NewWinProject > Open Folder in File Explorer > C:\Dev\NewProject\NewWinProject
 - .vs
 - NewWinProject.sln
 - NewWinProject.vcxproj
 - NewWinProject.vcsproj.filters
 - NewWinProject.vcxproj.user
 - Solution Explorer:
 - Solution 'NewWinProject' (1 of 1 project)

- NewWinProject
 - References (filters / NOT folders)
 - External Dependencies
 - Header Files
 - Resource Files
 - Source Files
- Add Folder: src (to hold all project/solutions files)
 - CLK > NewWinProject > Show All Files (icon)
 - RT+CLK > NewWinProject > Add > New Folder > src
- Add main.cpp in new folder: src
 - Select src > RT+CLK > Add > NewItem... > C++ File (.cpp)
 - Name: main.cpp
 - Location: C:\Dev\NewProject\NewWinProject\src > Add
- Check Project Filters: source files for new main.cpp
 - CLK > Show All Files (toggles back to filter folders)
 - main.cpp is now under source files
- Write quick “Hello World!” program in main.cpp
 - RT+CLK > NewWinProject > Build: 1 succeeded
 - Find project executable (NewWinProject.exe) file → located in Debug folder
- Change VS2019 Property Settings
 - RT+CLK > NewWinProject > Project > Properties (Alt + Enter)
 - Configurations: Active(Debug) → All Configurations
 - Platform: Active(Win32) → All Platforms
 - Configuration Properties > General
 - Output Directory: <different options> → \$(SolutionDir)bin\\$(Platform)\\$(Configuration)\ > Edit → Apply → Ok
 - Intermediate Directory: <different options> → \$(SolutionDir)bin\intermediates\\$(Platform)\\$(Configuration)\ > Edit → Apply → Ok
- Clean Project and Rebuild
 - RT+CLK > NewWinProject > Clean: 1 succeeded
 - Delete folders: Debug, bin
 - RTCLK > NewWinProject > Build (Rebuild)
 - Check new bin, bin\intermediates, bin\intermediates\Win32, bin\intermediates\Win32\Debug (7 files + 1 folder),
 - Check bin\Win32, bin\Win32\Debug (3 files with NewWinProject.exe)

OpenGL Shading Language

The **OpenGL Shading Language** (GLSL) is the principal shading language for OpenGL. While, thanks to [OpenGL Extensions](#), there are several shading languages available for use in OpenGL, GLSL (and [SPIR-V](#)) are supported directly by OpenGL without extensions.

GLSL is a C-style language. The language has undergone a number of version changes, and it shares the deprecation model of OpenGL. The current version of GLSL is 4.60. [OpenGL Shading Language - OpenGL Wiki \(khronos.org\)](#)

Standard library

The OpenGL Shading Language defines a number of **standard functions**. Some standard functions are specific to certain shader stages, while most are available in any stage. There is reference documentation for these functions available [here](#).

Variable types

Main article: [Data Type \(GLSL\)](#)

C has a number of basic types. GLSL uses some of these, but adds many more.

Type qualifiers

Main article: [Type Qualifier \(GLSL\)](#)

GLSL's uses a large number of qualifiers to specify where the values that various variables contain come from. Qualifiers also modify how those variables can be used.

Interface blocks

Main article: [Interface Block \(GLSL\)](#)

Certain variable definitions can be grouped into interface blocks. These can be used to make communication between different shader stages easier, or to allow storage for variables to come from a buffer object.

Predefined variables

Main article: [Built-in Variable \(GLSL\)](#)

The different shader stages have a number of predefined variables for them. These are provided by the system for various system-specific use.

Using GLSL shaders

Building shaders

Attributes and draw buffers

For the stages at the start and end of the pipeline (vertex and fragment, respectively), the initial input values and final output values do not come from or go to shader stages. The input values to a vertex shader come from [vertex data specified](#) in a [vertex array object](#), pulled from [vertex buffer objects](#) during [Vertex Rendering](#). The output values of a fragment shader are piped to particular buffers for the currently bound framebuffer; either the [default framebuffer](#) or a [framebuffer object](#).

Because of this, there is a mapping layer for the program's inputs and outputs. The vertex shader's input names are mapped to attribute indices, while the fragment shader's output names are mapped to draw buffer indices. This mapping can be created before the program is linked. If it is not, or if the mapping does not cover all of the inputs and outputs, then the linker will automatically define what indices are mapped to which unmapped input or output names. This auto-generated mapping can be queried by the user after the program is linked.

Setting uniforms

Main article: [GLSL Uniform#Uniform management](#)

Uniforms in GLSL are shader variables that are set from user code, but only are allowed to change between different `glDraw*` calls. Uniforms can be queried and set by the code external to a particular shader. Uniforms can be arranged into blocks, and the data storage for these blocks can come from buffer objects.

Setting samplers

Main article: [GLSL Sampler#Binding textures to samplers](#)

Samplers are special types which must be defined as uniforms. They represent bound textures in the OpenGL context. They are set like integer, 1D uniform values.

Error Checking

This piece of code shows the process of loading a vertex and fragment shaders. Then it compiles them and also checks for errors.

The idea here is to encourage newcomers to GLSL to always check for errors. It is in C++ but that doesn't matter.

Note that the process of loading and compiling shaders hasn't changed much over the different GL versions

Introduction

This guide will teach you the basics of using OpenGL to develop modern graphics applications. There are a lot of other guides on this topic, but there are some major points where this guide differs from those. We will not be discussing any of the old parts of the OpenGL specification.

That means you'll be taught how to implement things yourself, instead of using deprecated functions like `glBegin` and `glLight`.

Anything that is not directly related to OpenGL itself, like creating a window and loading textures from files, will be done using a few small libraries.

To show you how much it pays off to do things yourself, this guide also contains a lot of interactive examples to make it both fun and easy to learn all the different aspects of using a low-level graphics library like OpenGL!

As an added bonus, you always have the opportunity to ask questions at the end of each chapter in the comments section. I'll try to answer as many questions as possible, but always remember that there are plenty of people out there who are willing to help you with your issues. Make sure to help us help you by specifying your platform, compiler, the relevant code section, the result you expect and what is actually happening.

Window and OpenGL context

Before you can start drawing things, **you need to initialize OpenGL. This is done by creating an OpenGL context, which is essentially a state machine that stores all data related to the rendering of your application.** When your application closes, the OpenGL context is destroyed and everything is cleaned up.

The problem is that **creating a window and an OpenGL context is not part of the OpenGL specification.** That means it is done differently on every platform out there! Developing applications using OpenGL is all about being portable, so this is the last thing we need. Luckily there are libraries out there that abstract this process, so that you can maintain the same codebase for all supported platforms.

While the available libraries out there all have advantages and disadvantages, they do all have a certain program flow in common. You start by specifying the properties of the game window, such as the title and the size and the properties of the OpenGL context, like the anti-aliasing level. Your application will then initiate the event loop, which contains an important set of tasks that need to be completed over and over again until the window closes. These tasks usually handle window events like mouse clicks, updating the rendering state and then drawing.

```
#include <libraryheaders>

int main()
{
    createWindow(title, width, height);
    createOpenGLContext(settings);
    while (windowOpen)
    {
        while (event = newEvent())
            handleEvent(event);
        updateScene();
        drawGraphics();
        presentGraphics();
    }
    return 0;
}
```

When rendering a frame, the results will be stored in an offscreen buffer known as the **back buffer** to make sure the user only sees the final result. The `presentGraphics()` call will copy the result from the back buffer to the visible window buffer, the **front buffer**. Every application that makes use of real-time graphics will have a program flow that comes down to this, whether it uses a library or native code.

Supporting resizable windows with OpenGL introduces some complexities as resources need to be reloaded and buffers need to be recreated to fit the new window size. It's more convenient for the learning process to not bother with such details yet, so we'll only deal with fixed size (fullscreen) windows for now.

Libraries

There are many libraries around that can create a window and an accompanying OpenGL context for you. There is no best library out there, because everyone has different needs and ideals. I've chosen to discuss the process for the three most popular libraries here for completeness, but you can find more detailed guides on their respective websites. All code after this chapter will be independent of your choice of library here.

GLFW, as the name implies, is a C library specifically designed for use with OpenGL. Unlike SDL and SFML it only comes with the absolute necessities: window and context creation and input management. It offers the most control over the OpenGL context creation out of these three libraries.

GLFW

GLFW is tailored specifically for using OpenGL, so it is by far the easiest to use for our purpose.

Building

After you've downloaded the GLFW binaries package from the website or compiled the library yourself, you'll find the headers in the `include` folder and the libraries for your compiler in one of the `lib` folders.

- Add the appropriate `lib` folder to your library path and link with `GLFW`.
- Add the `include` folder to your include path.

You can also dynamically link with **GLFW** if you want to. Simply link with `GLFWDLL` and include the shared library with your executable.

Here is a simple snippet of code to check your build configuration:

```
#include <GLFW/glfw3.h>
#include <thread>

int main()
{
    glfwInit();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    glfwTerminate();
}
```

It should show a console application and exit after a second. If you run into any trouble, just ask in the comments and you'll receive help.

Code

Start by simply including the GLFW header and define the entry point of the application.

```
#include <GLFW/glfw3.h>

int main()
{
    return 0;
}
```

To use GLFW, it needs to be initialized when the program starts and you need to give it a chance to clean up when your program closes. The `glfwInit` and `glfwTerminate` functions are geared towards that purpose.

```
glfwInit();
...
glfwTerminate();
```


The next thing to do is creating and configuring the window. Before calling `glfwCreateWindow`, we first set some options.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

GLFWwindow* window = glfwCreateWindow(800, 600, "OpenGL", nullptr, nullptr); // Windowed
GLFWwindow* window = glfwCreateWindow(800, 600, "OpenGL", glfwGetPrimaryMonitor(), nullptr); // Fullscreen
```

You'll immediately notice the first three lines of code that are only relevant for this library. **It is specified that we require the OpenGL context to support OpenGL 3.2 at the least.** The `GLFW_OPENGL_PROFILE` option specifies that we want a context that only supports the new core functionality.

- The first two parameters of `glfwCreateWindow` specify the **width** and **height** of the drawing surface and
- The third parameter specifies the **window title**.
- The fourth parameter should be set to `NULL` for windowed mode and `glfwGetPrimaryMonitor()` for fullscreen mode.
- The last parameter allows you to specify an **existing OpenGL context** to share resources like textures with. The `glfwWindowHint` function is used to specify additional requirements for a window.

After creating the window, the OpenGL context has to be made active:

```
glfwMakeContextCurrent(window);
```

Next comes the **event loop**, which in the case of GLFW works a little differently than the other libraries. **GLFW uses a so-called closed event loop**, which means you only have to handle events when you need to. That means your event loop will look really simple:

```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

The only required functions in the loop are **glfwSwapBuffers** to swap the back buffer and front buffer after you've finished drawing and **glfwPollEvents** to retrieve window events. If you are making a fullscreen application, you should handle the escape key to easily return to the desktop.

```
if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    glfwSetWindowShouldClose(window, GL_TRUE);
```

One more thing

Unfortunately, we can't just call the functions we need yet. This is because it's the duty of the graphics card vendor to implement OpenGL functionality in their drivers based on what the graphics card supports. You wouldn't want your program to only be compatible with a single driver version and graphics card, so we'll have to do something clever.

Your program needs to check which functions are available at runtime and link with them dynamically. This is done by finding the addresses of the functions, assigning them to function pointers and calling them. That looks something like this:

Let me begin by asserting that it is perfectly normal to be scared by this snippet of code. You may not be familiar with the concept of function pointers yet, but at least try to roughly understand what is happening here. You can imagine that going through this process of defining prototypes and finding addresses of functions is very tedious and in the end nothing more than a complete waste of time.

The good news is that there are libraries that have solved this problem for us. The **most popular and best maintained library right now is GLEW** and there's no reason for that to change anytime soon. Nevertheless, the alternative library GLEE works almost completely the same save for the initialization and cleanup code.

If you haven't built GLEW yet, do so now. We'll now add GLEW to your project.

Start by linking your project with the static GLEW library in the lib folder. This is either **glew32s.lib or GLEW depending on your platform.**

Add the include folder to your include path.

Now just include the header in your program, but make sure that it is included before the OpenGL headers or the library you used to create your window.

```
#define GLEW_STATIC
#include <GL/glew.h>
```

Don't forget to define `GLEW_STATIC` either using this preprocessor directive or by adding the `-DGLEW_STATIC` directive to your compiler command-line parameters or project settings.

If you prefer to dynamically link with GLEW, leave out the define and link with `glew32.lib` instead of `glew32s.lib` on Windows. Don't forget to include `glew32.dll` or `libGLEW.so` with your executable!

Now all that's left is calling `glewInit()` after the creation of your window and OpenGL context. The `glewExperimental` line is necessary to force GLEW to use a modern OpenGL method for checking if a function is available.

```
glewExperimental = GL_TRUE;  
glewInit();
```

Make sure that you've set up your project correctly by calling the `glGenBuffers` function, which was loaded by GLEW for you!

```
GLuint vertexBuffer;  
glGenBuffers(1, &vertexBuffer);  
  
printf("%u\n", vertexBuffer);
```

Your program should compile and run without issues and display the number `1` in your console. If you need more help with using GLEW, you can refer to the [website](#) or ask in the comments.

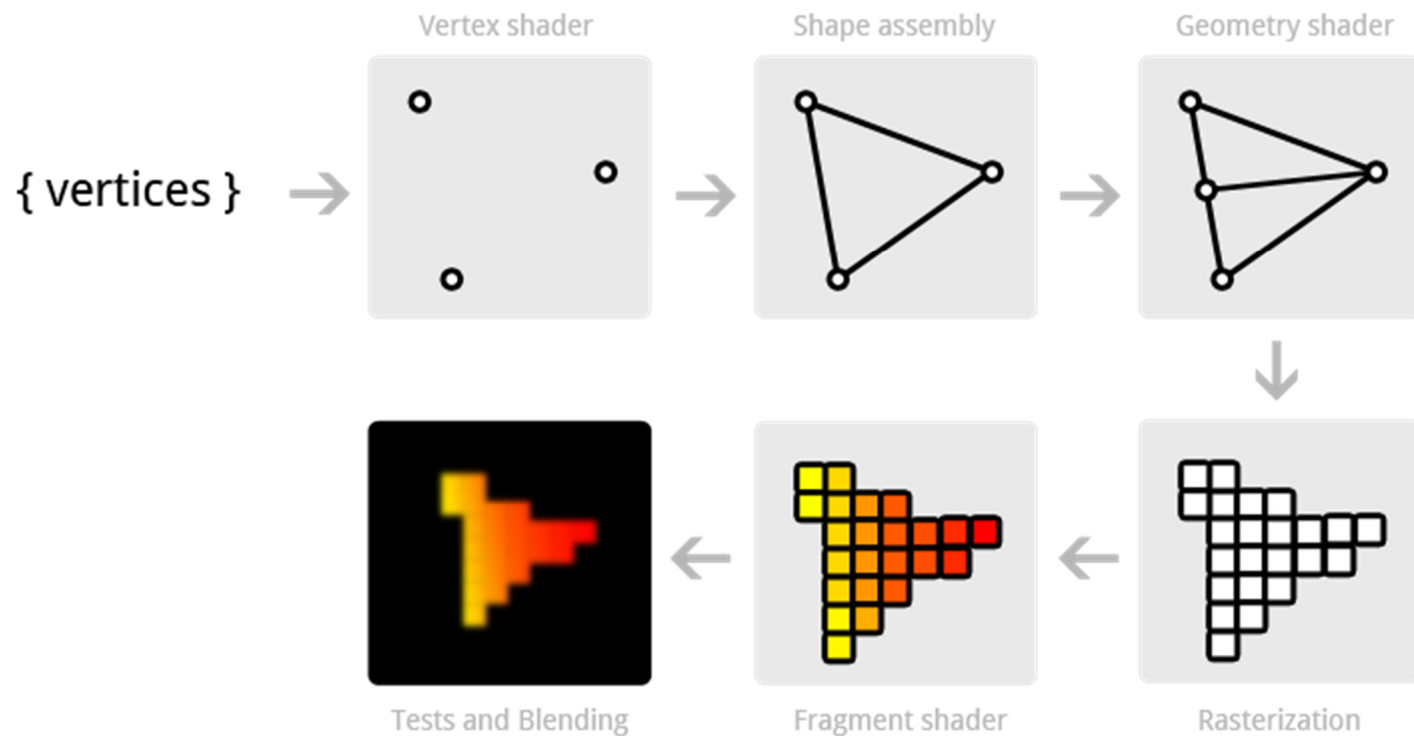
Now that we're past all of the configuration and initialization work, I'd advise you to make a copy of your current project so that you won't have to write all of the boilerplate code again when starting a new project.

Now, let's get to [drawing things](#)!

The graphics pipeline

By learning OpenGL, you've decided that you want to do all of the hard work yourself. That inevitably means that you'll be thrown in the deep, but once you understand the essentials, you'll see that doing things *the hard way* doesn't have to be so difficult after all. To top that all, the exercises at the end of this chapter will show you the sheer amount of control you have over the rendering process by doing things the modern way!

The **graphics pipeline** covers all of the steps that follow each other up on processing the input data to get to the final output image. I'll explain these steps with help of the following illustration.



It all begins with the **vertices**, these are the points from which shapes like triangles will later be constructed. Each of these points is stored with certain attributes and it's up to you to decide what kind of attributes you want to store. Commonly used attributes are 3D position in the world and texture coordinates.

The **vertex shader** is a small program running on your graphics card that processes every one of these input vertices individually. This is where the perspective transformation takes place, **which projects vertices with a 3D world position onto your 2D screen!** It also passes important attributes like color and texture coordinates further down the pipeline.

After the input vertices have been transformed, the graphics card will form **triangles, lines or points** out of them. These shapes are called **primitives** because they form the basis of more complex shapes. There are some additional drawing modes to choose from, like triangle strips and line strips. These reduce the number of vertices you need to pass if you want to create objects where each next primitive is connected to the last one, like a continuous line consisting of several segments.

After the final list of shapes is composed and converted to screen coordinates, the **rasterizer** turns the visible parts of the shapes into **pixel-sized fragments**. The **vertex attributes** coming from the **vertex shader** or **geometry shader** are **interpolated and passed as input to the fragment shader for each fragment**. As you can see in the image, the colors are smoothly **interpolated** over the fragments that make up the triangle, even though only 3 points were specified.

The **fragment shader** processes each individual fragment along with its interpolated attributes and should output the final color. This is usually done by **sampling from a texture** using the **interpolated texture coordinate vertex attributes** or simply outputting a color. In more advanced scenarios, there could also be calculations related to lighting and shadowing and special effects in this program. **The shader also has the ability to discard certain fragments, which means that a shape will be see-through there.**

Finally, the end result is composed from all these **shape fragments** by **blending** them together and performing depth and **stencil testing**. All you need to know about these last two right now, is that they allow you to use additional rules to throw away certain fragments and let others pass. For example, if one triangle is obscured by another triangle, the fragment of the closer triangle should end up on the screen.

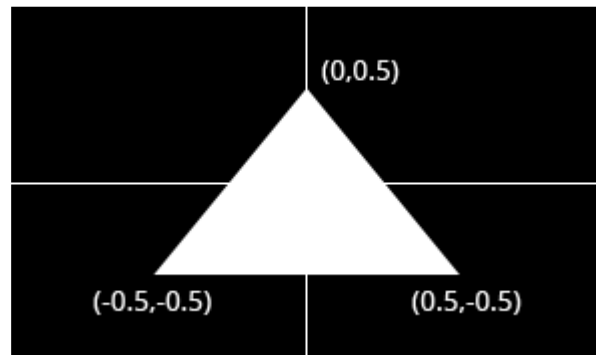
Now that you know how your graphics card turns an array of vertices into an image on the screen, let's get to work!

Vertex input

The first thing you have to decide on is what data the graphics card is going to need to draw your scene correctly. As mentioned above, this data comes in the form of vertex attributes. You're free to come up with any kind of attribute you want, but it all inevitably begins with the **world position**. Whether you're doing 2D graphics or 3D graphics, this is the attribute that will determine where the objects and shapes end up on your screen in the end.

Device coordinates

When your vertices have been processed by the pipeline outlined above, their coordinates will have been transformed into *device coordinates*. Device X and Y coordinates are mapped to the screen between -1 and 1.



Just like a graph, the center has coordinates $(0, 0)$ and the y axis is positive above the center. This seems unnatural because graphics applications usually have $(0, 0)$ in the top-left corner and $(width, height)$ in the bottom-right corner, but it's an excellent way to simplify 3D calculations and to stay resolution independent.

The triangle above consists of 3 vertices positioned at $(0, 0.5)$, $(0.5, -0.5)$ and $(-0.5, -0.5)$ in clockwise order. It is clear that the only variation between the vertices here is the **position**, so that's the only attribute we need. Since we're passing the device coordinates directly, an X and Y coordinate suffices for the position.

OpenGL expects you to send all of your vertices in a single array, which may be confusing at first. To understand the format of this array, let's see what it would look like for our triangle.

```
float vertices[] = {  
    0.0f, 0.5f, // Vertex 1 (X, Y)  
    0.5f, -0.5f, // Vertex 2 (X, Y)  
    -0.5f, -0.5f // Vertex 3 (X, Y)  
};
```

As you can see, this array should simply be a list of all vertices with their attributes packed together. The order in which the attributes appear doesn't matter, as long as it's the same for each vertex.

The order of the vertices doesn't have to be sequential (i.e. the order in which shapes are formed), but this requires us to provide extra data in the form of an **element buffer.** This will be discussed at the end of this chapter as it would just complicate things for now.

The next step is to upload this vertex data to the graphics card. This is important because the memory on your graphics card is much faster *and* you won't have to send the data again every time your scene needs to be rendered (about 60 times per second).

This is done by creating a **Vertex Buffer Object (VBO)**:

```
GLuint vbo;  
glGenBuffers(1, &vbo); // Generate 1 buffer
```

The **graphics memory** is managed by OpenGL, so instead of a pointer you get a positive number as a reference to it. **GLuint** is simply a cross-platform substitute for **unsigned int**, just like **GLint** is one for **int**. You will need this number to make the VBO active and to destroy it when you're done with it.

To upload the actual data to it you first have to make it the active object by calling **glBindBuffer** to bind the **Vertex Buffer Object (vbo)** to the active buffer type.

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

As hinted by the **GL_ARRAY_BUFFER** enum value there are other types of buffers, but they are not important right now. **This statement makes the VBO we just created the active array buffer**. Now that it's active we can copy the vertex data to it.

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Notice that this function doesn't refer to the id of our VBO, but instead to the active array buffer. The second parameter specifies the **size in bytes**. The final parameter is very important and its value depends on the **usage** of the vertex data. I'll outline the ones related to drawing here:

- **GL_STATIC_DRAW**: The vertex data will be uploaded once and drawn many times (e.g. the world).
- **GL_DYNAMIC_DRAW**: The vertex data will be created once, changed from time to time, but drawn many times more than that.
- **GL_STREAM_DRAW**: The vertex data will be uploaded once and drawn once.

This usage value will determine in what kind of memory the data is stored on your graphics card for the highest efficiency. For example, VBOs with **GL_STREAM_DRAW** as type may store their data in memory that allows faster writing in favor of slightly slower drawing.

The vertices with their attributes have been copied to the graphics card now, but they're not quite ready to be used yet.

Remember that we can make up any kind of attribute we want and in any order, so now comes the moment where you have to explain to the graphics card how to handle these attributes. This is where you'll see how flexible modern OpenGL really is.

Shaders

As discussed earlier, **there are three shader stages your vertex data will pass through**. Each shader stage has a strictly defined purpose and in older versions of OpenGL, you could only slightly tweak what happened and how it happened. With modern OpenGL, it's up to us to instruct the graphics card what to do with the data. This is why it's possible to decide per application what attributes each vertex should have. You'll have to implement both the vertex and fragment shader to get something on the screen, the geometry shader is optional and is discussed later.

Shaders are written in a C-style language called GLSL (OpenGL Shading Language). OpenGL will compile your program from source at runtime and copy it to the graphics card. Each version of OpenGL has its own version of the shader language with availability of a certain feature set and we will be using GLSL 1.50. This version number may seem a bit off when we're using OpenGL 3.2, but that's because shaders were only introduced in OpenGL 2.0 as GLSL 1.10. Starting from OpenGL 3.3, this problem was solved and the GLSL version is the same as the OpenGL version.

Vertex shader

The **vertex shader** is a program on the graphics card that processes each vertex and its attributes as they appear in the **vertex array**. Its duty is to output the final vertex position in device coordinates and to output any data the fragment shader requires. That's why the **3D transformation** should take place here. The fragment shader depends on attributes like the color and texture coordinates, which will usually be passed from input to output without any calculations.

Remember that our vertex position is already specified as device coordinates and no other attributes exist, so the vertex shader will be fairly bare bones.

```
#version 150 core
in vec2 position;
void main()
{
    gl_Position = vec4(position, 0.0, 1.0);
}
```


The `#version` **preprocessor directive** is used to indicate that the code that follows is GLSL 1.50 code using OpenGL's core profile. Next, we specify that there is only **one attribute, the position**. **Apart from the regular C types, GLSL has built-in vector and matrix types identified by `vec*` and `mat*` identifiers.** The type of the values within these constructs is always a `float`. The number after `vec` specifies the number of components (x, y, z, w) and the number after `mat` specifies the number of rows /columns. Since the position attribute consists of only an X and Y coordinate, `vec2` is perfect.

You can be quite creative when working with these vertex types. In the example above a shortcut was used to set the first two components of the `vec4` to those of `vec2`. These two lines are equal:

```
gl_Position = vec4(position, 0.0, 1.0);  
gl_Position = vec4(position.x, position.y, 0.0, 1.0);
```

When you're working with colors, you can also access the individual components with `r`, `g`, `b` and `a` instead of `x`, `y`, `z` and `w`. This makes no difference and can help with clarity.

The final position of the vertex is assigned to the special `gl_Position` variable, because the position is needed for primitive assembly and many other built-in processes. For these to function correctly, the last value `w` needs to have a value of `1.0f`. Other than that, you're free to do anything you want with the attributes and we'll see how to output those when we add color to the triangle later in this chapter.

Fragment shader

The output from the vertex shader is interpolated over all the pixels on the screen covered by a primitive. These pixels are called fragments and this is what the fragment shader operates on. Just like the vertex shader it has one mandatory output, the final color of a fragment. It's up to you to write the code for computing this color from vertex colors, texture coordinates and any other data coming from the vertex shader.

Our triangle only consists of white pixels, so the fragment shader simply outputs that color every time:

```
#version 150 core  
out vec4 outColor;  
  
void main()  
{  
    outColor = vec4(1.0, 1.0, 1.0, 1.0);  
}
```

You'll immediately notice that we're **not** using some built-in variable for outputting the color, say `gl_FragColor`. This is because a fragment shader can in fact output multiple colors and we'll see how to handle this when actually loading these shaders. The `outColor` variable uses the type `vec4`, because each color consists of a red, green, blue and alpha component. Colors in OpenGL are generally represented as floating point numbers between `0.0` and `1.0` instead of the common `0` and `255`.

Compiling shaders

Compiling shaders is easy once you have loaded the source code (either from file or as a hard-coded string). You can easily include your shader source in the C++ code through C++11 raw string literals:

```
const char* vertexSource = R"glsl(
    #version 150 core
    in vec2 position;

    void main()
    {
        gl_Position = vec4(position, 0.0, 1.0);
    }
)glsl";
```

Just like vertex buffers, creating a shader itself starts with creating a shader object and loading data into it.

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexSource, NULL);
```

Unlike VBOs, you can simply pass a reference to shader functions instead of making it active or anything like that. The `glShaderSource` function can take multiple source strings in an array, but you'll usually have your source code in one char array. The last parameter can contain an array of source code string lengths, passing NULL simply makes it stop at the null terminator.

All that's left is compiling the shader into code that can be executed by the graphics card now:

```
glCompileShader(vertexShader);
```

Be aware that if the shader fails to compile, e.g. because of a syntax error, `glGetError` will not report an error! See the block below for info on how to debug shaders.

Checking if a shader compiled successfully

```
GLint status;  
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &status);  
If status is equal to GL_TRUE, then your shader was compiled successfully.
```

Retrieving the compile log

```
char buffer[512];  
glGetShaderInfoLog(vertexShader, 512, NULL, buffer);
```

This will store the first 511 bytes + null terminator of the compile log in the specified buffer. The log may also report useful warnings even when compiling was successful, so it's useful to check it out from time to time when you develop your shaders.

The fragment shader is compiled in exactly the same way:

```
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);  
glShaderSource(fragmentShader, 1, &fragmentSource, NULL);  
glCompileShader(fragmentShader);
```

Again, be sure to check if your shader was compiled successfully, because it will save you from a headache later on.

Combining shaders into a program

Up until now the vertex and fragment shaders have been two separate objects. While they've been programmed to work together, they aren't actually connected yet. This connection is made by creating a *program* out of these two shaders.

```
GLuint shaderProgram = glCreateProgram();  
glAttachShader(shaderProgram, vertexShader);  
glAttachShader(shaderProgram, fragmentShader);
```

Since a **fragment shader** is allowed to write to **multiple framebuffers**, you need to explicitly specify which output is written to which framebuffer. This needs to happen before linking the program. However, since this is 0 by default and there's only one output right now, the following line of code is not necessary:

```
glBindFragDataLocation(shaderProgram, 0, "outColor");
```

Use `glDrawBuffers` when rendering to multiple framebuffers, because only the first output will be enabled by default.

After attaching both the fragment and vertex shaders, the connection is made by *linking* the program. It is allowed to make changes to the shaders after they've been added to a program (or multiple programs!), but the actual result will not change until a program has been linked again. It is also possible to attach multiple shaders for the same stage (e.g. fragment) if they're parts forming the whole shader together. A shader object can be deleted with `glDeleteShader`, but it will not actually be removed before it has been detached from all programs with `glDetachShader`.

```
glLinkProgram(shaderProgram);
```

Just like a vertex buffer, only one program can be active at a time.

Making the link between vertex data and attributes

Although we have our vertex data and shaders now, OpenGL still doesn't know how the attributes are formatted and ordered. You first need to retrieve a reference to the `position` input in the vertex shader:

```
GLuint posAttrib = glGetAttribLocation(shaderProgram, "position");
```

The `location` is a number depending on the order of the input definitions. The first and only input `position` in this example will always have location 0.

With the reference to the input, you can specify how the data for that input is retrieved from the array:

```
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

- The first parameter references the input.
- The second parameter specifies the number of values for that input, which is the same as the number of components of the `vec`.
- The third parameter specifies the type of each component and
- The fourth parameter specifies whether the input values should be normalized between -1
`.0` and `1.0` (or `0.0` and `1.0` depending on the format) if they aren't floating point numbers.

The last two parameters are arguably the most important here as they specify how the attribute is laid out in the vertex array.

- The first number specifies the *stride*, or how many bytes are between each position attribute in the array. The value 0 means that there is no data in between. This is currently the case as the position of each vertex is immediately followed by the position of the next vertex.
- The last parameter specifies the *offset*, or how many bytes from the start of the array the attribute occurs. Since there are no other attributes, this is 0 as well.

It is important to know that this function will store not only the stride and the offset, but also the VBO that is currently bound to `GL_ARRAY_BUFFER`. That means that you don't have to explicitly bind the correct VBO when the actual drawing functions are called. This also implies that you can use a different VBO for each attribute.

Don't worry if you don't fully understand this yet, as we'll see how to alter this to add more attributes soon enough.

```
glEnableVertexAttribArray(posAttrib);
```

Last, but not least, the vertex attribute array needs to be enabled.

Vertex Array Objects

You can imagine that real graphics programs use many different shaders and vertex layouts to take care of a wide variety of needs and special effects. Changing the active shader program is easy enough with a call to `glUseProgram`, but it would be quite inconvenient if you had to set up all of the attributes again every time.

Luckily, OpenGL solves that problem with **Vertex Array Objects (VAO)**. VAOs store all of the links between the attributes and your VBOs with raw vertex data.

A VAO is created in the same way as a VBO:

```
GLuint vao;  
glGenVertexArrays(1, &vao);
```

To start using it, simply bind it:

```
glBindVertexArray(vao);
```

As soon as you've bound a certain VAO, every time you call `glVertexAttribPointer`, that information will be stored in that VAO.

This makes switching between different vertex data and vertex formats as easy as binding a different VAO! Just remember that a VAO doesn't store any vertex data by itself, it just references the VBOs you've created and how to retrieve the attribute values from them.

Since only calls after binding a VAO stick to it, make sure that you've created and bound the VAO at the start of your program. Any vertex buffers and element buffers bound before it will be ignored.

Drawing

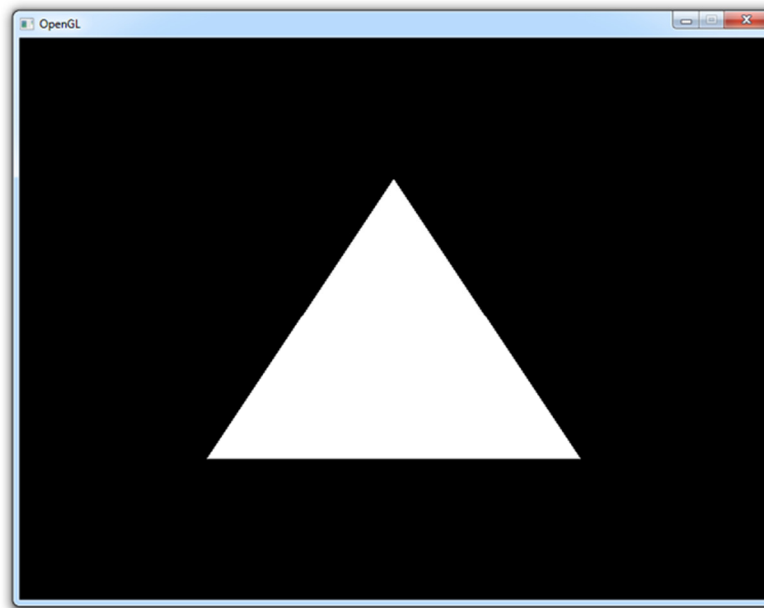
Now that you've loaded the vertex data, created the shader programs and linked the data to the attributes, you're ready to draw the triangle. The VAO that was used to store the attribute information is already bound, so you don't have to worry about that.

All that's left is to simply call `glDrawArrays` in your main loop:

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- The first parameter specifies the kind of primitive (commonly `point`, `line` or `triangle`),
- The second parameter specifies `how many vertices to skip at the beginning and`
- The last parameter specifies the `number of vertices` (not primitives!) `to process`.

When you run your program now, you should see the following:



If you don't see anything, make sure that the shaders have compiled correctly, that the program has **linked correctly**, that the attribute array has been enabled, that the VAO has been bound before specifying the attributes, that your vertex data is correct and that `glGetError` returns `0`.

If you can't find the problem, try comparing your code to [this sample](#).

Uniforms

Right now, the white color of the triangle has been hard-coded into the shader code, but what if you wanted to change it after compiling the shader?

As it turns out, **vertex attributes are not the only way to pass data to shader programs**.

There is another way to pass data to the shaders called `uniforms`. These are essentially global variables, having the same value for all vertices and/or fragments.

To demonstrate how to use these, let's make it possible to change the color of the triangle from the program itself.

By making the color in the fragment shader a uniform, it will end up looking like this:

```
#version 150 core
uniform vec3 triangleColor;
out vec4 outColor;
void main()
{
    outColor = vec4(triangleColor, 1.0);
}
```

The last component of the output color is transparency, which is not very interesting right now. If you run your program now, you'll see that the triangle is black, because the value of `triangleColor` hasn't been set yet.

Changing the value of a uniform is just like setting vertex attributes, you first have to grab the location:

```
GLint uniColor = glGetUniformLocation(shaderProgram, "triangleColor");
```

The values of uniforms are changed with any of the `glUniformXY` functions, where X is the number of components and Y is the type. Common types are `f` (float), `d` (double) and `i` (integer).

```
glUniform3f(uniColor, 1.0f, 0.0f, 0.0f);
```

If you run your program now, you'll see that the triangle is **red**. To make things a little more exciting, try varying the color with the time by doing something like this in your main loop:

```
auto t_start = std::chrono::high_resolution_clock::now();
...
auto t_now = std::chrono::high_resolution_clock::now();
float time = std::chrono::duration_cast<std::chrono::duration<float>>(t_now - t_start).count();
glUniform3f(uniColor, (sin(time * 4.0f) + 1.0f) / 2.0f, 0.0f, 0.0f);
```

Although this example may not be very exciting, it does demonstrate that **uniforms are essential for controlling the behavior of shaders at runtime**.

Vertex attributes on the other hand are ideal for describing a **single vertex**.

Adding some more colors

Although uniforms have their place, **color is something we'd rather like to specify per corner of the triangle!**

Let's add a color attribute to the vertices to accomplish this.

We'll first have to add the extra attributes to the vertex data. Transparency IS NOT really relevant, so we'll only add the red, green and blue components:

```
float vertices[] = {  
    0.0f, 0.5f, 1.0f, 0.0f, 0.0f, // Vertex 1: Red  
    0.5f, -0.5f, 0.0f, 1.0f, 0.0f, // Vertex 2: Green  
    -0.5f, -0.5f, 0.0f, 0.0f, 1.0f // Vertex 3: Blue  
};
```

Then we have to change the vertex shader to take it as input and pass it to the fragment shader:

```
#version 150 core  
  
in vec2 position;  
in vec3 color;  
  
out vec3 Color;  
  
void main()  
{  
    Color = color;  
    gl_Position = vec4(position, 0.0, 1.0);  
}
```

And `Color` is added as input to the fragment shader:

```
#version 150 core

in vec3 Color;
out vec4 outColor;

void main()
{
    outColor = vec4(Color, 1.0);
}
```

Make sure that the output of the vertex shader and the input of the fragment shader have the same name, or the shaders will not be linked properly.

Now, we just need to alter the attribute pointer code a bit to accommodate for the new `X, Y, R, G, B` attribute order.

```
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 5*sizeof(float), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 5*sizeof(float), (void*)(2*sizeof(float)));
```

The fifth parameter is set to `5*sizeof(float)` now, because each vertex consists of 5 floating point attribute values.

The sixth parameter offset of `2*sizeof(float)` for the color attribute is there because each vertex starts with 2 floating point values for the position that it has to skip over.

And we're done!

Textures objects and parameters

Just like **VBOs** and **VAOs**, **textures are objects** that need to be generated first by calling a function. It shouldn't be a surprise at this point what this function is called.

```
GLuint tex;  
glGenTextures(1, &tex);
```

Textures are typically used for images to decorate 3D models, but in reality, they can be used to store many different kinds of data.

It's possible to have **1D**, **2D** and even **3D** textures, which can be used to store bulk data on the GPU.

An example of another use for textures is **storing terrain information**.

This article will pay attention to the use of textures for images, but the principles generally apply to all kinds of textures.

```
glBindTexture(GL_TEXTURE_2D, tex);
```

Just like other objects, textures have to be bound to apply operations to them.

Since images are 2D arrays of pixels, it will be bound to the **GL_TEXTURE_2D** target.

The pixels in the texture will be addressed using texture coordinates during drawing operations.

These coordinates range from **0.0** to **1.0** where **(0,0)** is conventionally the bottom-left corner and **(1,1)** is the top-right corner of the texture image.

The operation that uses these texture coordinates to retrieve color information from the pixels is called *sampling*.

There are different ways to approach this problem, each being appropriate for different scenarios.

OpenGL offers you many options to control how this sampling is done, of which the common ones will be discussed here.

Wrapping

The first thing you'll have to consider is how the texture should be sampled when a coordinate outside the range of 0 to 1 is given. OpenGL offers 4 ways of handling this:

- `GL_REPEAT`: The integer part of the coordinate will be ignored and a repeating pattern is formed.
- `GL_MIRRORED_REPEAT`: The texture will also be repeated, but it will be mirrored when the integer part of the coordinate is odd.
- `GL_CLAMP_TO_EDGE`: The coordinate will simply be clamped between 0 and 1.
- `GL_CLAMP_TO_BORDER`: The coordinates that fall outside the range will be given a specified border color.

These explanations may still be a bit cryptic and since OpenGL is all about graphics, let's see what all of these cases actually look like:



GL_REPEAT



GL_MIRRORED_REPEAT



GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER

The clamping can be set per coordinate, where the equivalent of `(x, y, z)` in texture coordinates is called `(s, t, r)`.

Texture parameter are changed with the `glTexParameter*` functions as demonstrated here

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

As before, the `i` here indicates the type of the value you want to specify.

If you use `GL_CLAMP_TO_BORDER` and you want to change the border color, you need to change the value of `GL_TEXTURE_BORDER_COLOR` by passing an RGBA float array:

```
float color[] = { 1.0f, 0.0f, 0.0f, 1.0f };  
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, color);
```

This operation will set the border color to red.

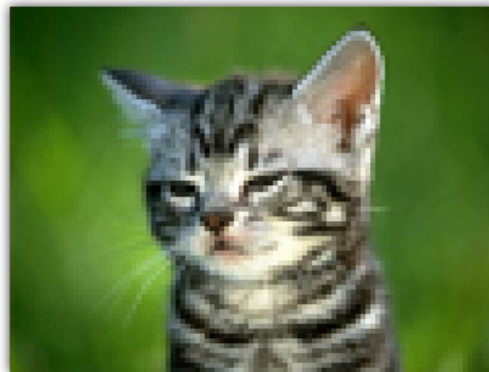
Filtering

Since texture coordinates are resolution independent, they won't always match a pixel exactly. This happens when a texture image is stretched beyond its original size or when it's sized down. OpenGL offers various methods to decide on the sampled color when this happens.

This process is called **filtering** and the following methods are available:

- `GL_NEAREST` : Returns the pixel that is closest to the coordinates.
- `GL_LINEAR` : Returns the weighted average of the 4 pixels surrounding the given coordinates.
- `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_LINEAR` :
Sample from mipmaps instead.

Before discussing mipmaps, let's first see the difference between **nearest** and **linear interpolation**. The original image is 16 times smaller than the rectangle it was rasterized on.



GL_NEAREST



GL_LINEAR

While **linear interpolation** gives a smoother result, it isn't always the most ideal option.

Nearest neighbor interpolation is more suited in games that want to mimic 8-bit graphics, because of the pixelated look.

You can specify which kind of interpolation should be used for two separate cases: scaling the image down and scaling the image up.

These two cases are identified by the keywords **GL_TEXTURE_MIN_FILTER** and **GL_TEXTURE_MAG_FILTER**.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

As you've seen, there is another way to filter textures: **mipmaps**.

Mipmaps are smaller copies of your texture that have been sized down and filtered in advance.

It is recommended that you use them because they result in both a higher quality and higher performance.

```
glGenerateMipmap(GL_TEXTURE_2D);
```

Generating them is as simple as calling the function above, so there's no excuse for not using them!

Note that you **DO** have to load the texture image itself **BEFORE** mipmaps can be generated from it.

To use mipmaps, select one of the four mipmap filtering methods.

- **GL_NEAREST_MIPMAP_NEAREST** : Uses the mipmap that most closely matches the size of the pixel being textured and samples with nearest neighbor interpolation.
- **GL_LINEAR_MIPMAP_NEAREST** : Samples the closest mipmap with linear interpolation.
- **GL_NEAREST_MIPMAP_LINEAR** : Uses the two mipmaps that most closely match the size of the pixel being textured and samples with nearest neighbor interpolation.
- **GL_LINEAR_MIPMAP_LINEAR** : Samples closest two mipmaps with linear interpolation.

There are some other texture parameters available, but they're suited for specialized operations.

You can read about them in the specification.

Loading texture images

Now that the texture object has been configured it's time to load the texture image. This is done by simply loading an array of pixels into it:

```
// Black/white checkerboard
float pixels[] = {
    0.0f, 0.0f, 0.0f,  1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 1.0f,  0.0f, 0.0f, 0.0f
};

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 2, 2, 0, GL_RGB, GL_FLOAT, pixels);
```

- The first parameter after the texture target is the *level-of-detail*, where `0` is the base image. This parameter can be used to load your own mipmap images.
- The second parameter specifies the *internal pixel format*, the format in which pixels should be stored on the graphics card. Many different formats are available, including compressed formats, so it's certainly worth taking a look at all of the options.
- The third and fourth parameters specify the *width and height of the image*.
- The fifth parameter should always have a value of `0` per the *specification*.
- The next two parameters describe the *format of the pixels in the array that will be loaded* and
- The final parameter specifies the *array itself*. The function begins loading the image at coordinate `(0, 0)`, so pay attention to this.

But how is the pixel array itself established? Textures in graphics applications will usually be a lot more sophisticated than simple patterns and will be loaded from files. Best practice is to have your files in a format that is natively supported by the hardware, but it may sometimes be more convenient to load textures from common image formats like JPG and PNG.

Unfortunately, **OpenGL DOES NOT** offer any helper functions to load pixels from these image files, but that's where third-party libraries come in handy again! The **SOIL library** will be discussed here along with some of the alternatives.

SOIL

SOIL (Simple OpenGL Image Library) is a small and easy-to-use library that loads image files directly into texture objects or creates them for you. You can start using it in your project by linking with `SOIL` and adding the `src` directory to your include path. It includes Visual Studio project files to compile it yourself.

Although SOIL includes functions to automatically create a texture from an image, it uses features that **ARE NOT** available in modern OpenGL. **Because of this we'll simply use SOIL as image loader and create the texture ourselves.**

```
int width, height;  
unsigned char* image = SOIL_load_image("img.png", &width, &height, 0, SOIL_LOAD_RGB);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
```

You can start configuring the texture parameters and generating mipmaps after this.

```
SOIL_free_image_data(image);
```

You can clean up the image data right after you've loaded it into the texture.

As mentioned before, OpenGL expects the first pixel to be located in the bottom-left corner, which means that textures will be flipped when loaded with SOIL directly. To counteract that, the code in the tutorial will use flipped Y coordinates for texture coordinates from now on. That means that 0, 0 will be assumed to be the top-left corner instead of the bottom-left. This practice might make texture coordinates more intuitive as a side-effect.

Alternative options

Other libraries that support a wide range of file types like SOIL are [DevIL](#) and [FreeImage](#). If you're just interested in one file type, it's also possible to use libraries like [libpng](#) and [libjpeg](#) directly. If you're looking for more of an adventure, have a look at the specification of the [BMP](#) and [TGA](#) file formats, it's not that hard to implement a loader for them yourself.

Using a texture

As you've seen, textures are sampled using texture coordinates and you'll have to add these as attributes to your vertices. Let's modify the last sample from the previous chapter to include these texture coordinates. **The new vertex array will now include the `s` and `t` coordinates for each vertex:**

```
float vertices[] = {  
    // Position(2), Color(3), Texcoords(2) for (4) Vertices  
    -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top-left  
    0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-right  
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // Bottom-right  
    -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // Bottom-left  
};
```


The **vertex shader** needs to be modified so that the texture coordinates are interpolated over the fragments:

```
...
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
...
void main()
{
    Texcoord = texcoord;
```

Just like when the color attribute was added, the attribute pointers need to be adapted to the new format:

```
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(float), 0);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(float), (void*)(2*sizeof(float)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
glEnableVertexAttribArray(texAttrib);
glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 7 * sizeof(float), (void*)(5*sizeof(float)));
```

As **two floats** were added for the **coordinates**, **one vertex** is now **7 floats in size** and the texture coordinate attribute consists of 2 of those floats.

Now just one thing remains: providing access to the texture in the fragment shader to sample pixels from it. This is done by adding a uniform of type **sampler2D**, which will have a **default value of 0**. This only needs to be changed when access has to be provided to multiple textures, which will be considered in the next section.

For this sample, the image of the kitten used above will be loaded using the SOIL library. Make sure that it is located in the working directory of the application.

```
int width, height;
unsigned char* image = SOIL_load_image("sample.png", &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);

SOIL_free_image_data(image);
```

To sample a pixel from a 2D texture using the sampler, the function `texture` can be called with the relevant sampler and texture coordinate as parameters.

We'll also multiply the sampled color with the color attribute to get an interesting effect. Your fragment shader will now look like this:

```
#version 150 core

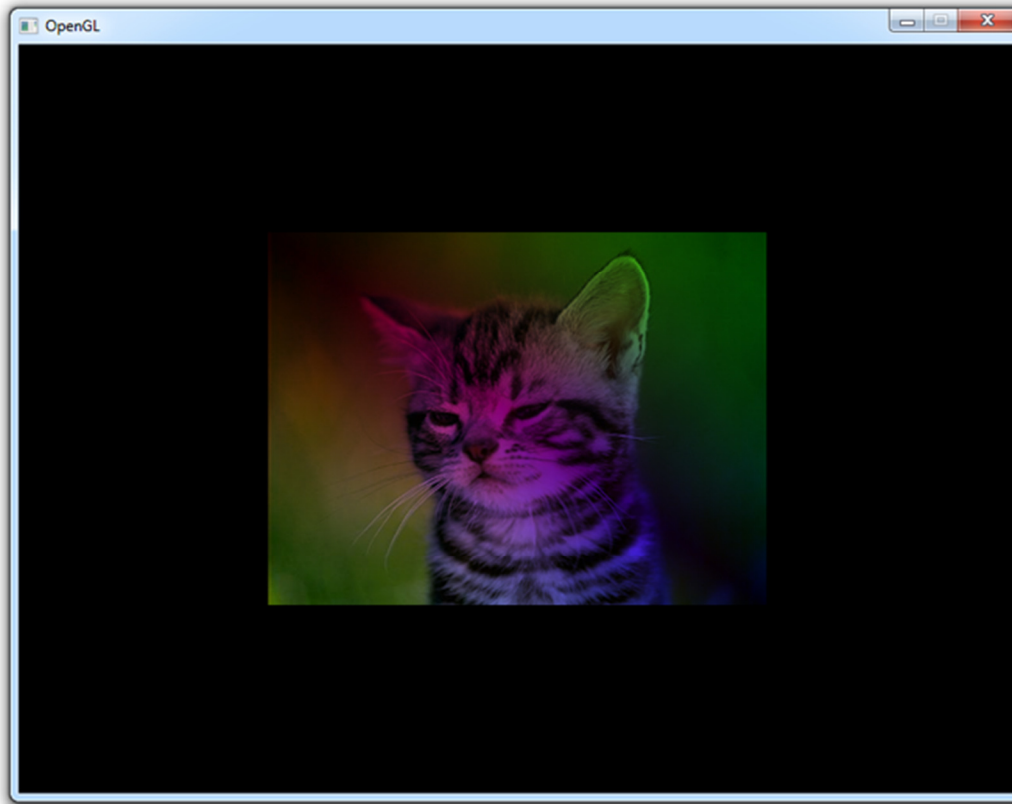
in vec3 Color;
in vec2 Texcoord;

out vec4 outColor;

uniform sampler2D tex;

void main()
{
    outColor = texture(tex, Texcoord) * vec4(Color, 1.0);
}
```

When running this application, you should get the following result:



If you get a black screen, make sure that your shaders compiled successfully and that the image is correctly loaded. If you can't find the problem, try comparing your code to the sample code.

Texture units

The sampler in your fragment shader is bound to texture unit `0`.

- Texture units are references to texture objects that can be sampled in a shader.
- Textures are bound to texture units using the `glBindTexture` function you've used before. Because you didn't explicitly specify which texture unit to use, the texture was bound to `GL_TEXTURE0`.
- That's why the default value of `0` for the sampler in your shader worked fine.

The function `glActiveTexture` specifies which texture unit a texture object is bound to when `glBindTexture` is called

```
glActiveTexture(GL_TEXTURE0);
```

The amount of texture units supported differs per graphics card, but it will be at least 48. It is safe to say that you will never hit this limit in even the most extreme graphics applications.

To practice with sampling from multiple textures, let's try blending the images of the kitten and one of a puppy to get the best of both worlds!

Let's first modify the fragment shader to sample from two textures and blend the pixels:

```
...  
  
uniform sampler2D texKitten;  
uniform sampler2D texPuppy;  
  
void main()  
{  
    vec4 colKitten = texture(texKitten, Texcoord);  
    vec4 colPuppy = texture(texPuppy, Texcoord);  
    outColor = mix(colKitten, colPuppy, 0.5);  
}
```

The `mix` function here is a special GLSL function that linearly interpolates between two variables based on the third parameter.

A value of `0.0` will result in the first value, a value of `1.0` will result in the second value and a value in between will result in a mixture of both values.

You'll have the chance to experiment with this in the exercises.

Now that the two samplers are ready, you'll have to assign the first two texture units to them and bind the two textures to those units.

This is done by adding the proper `glActiveTexture` calls to the texture loading code.

```
GLuint textures[2];
glGenTextures(2, textures);

int width, height;
unsigned char* image;

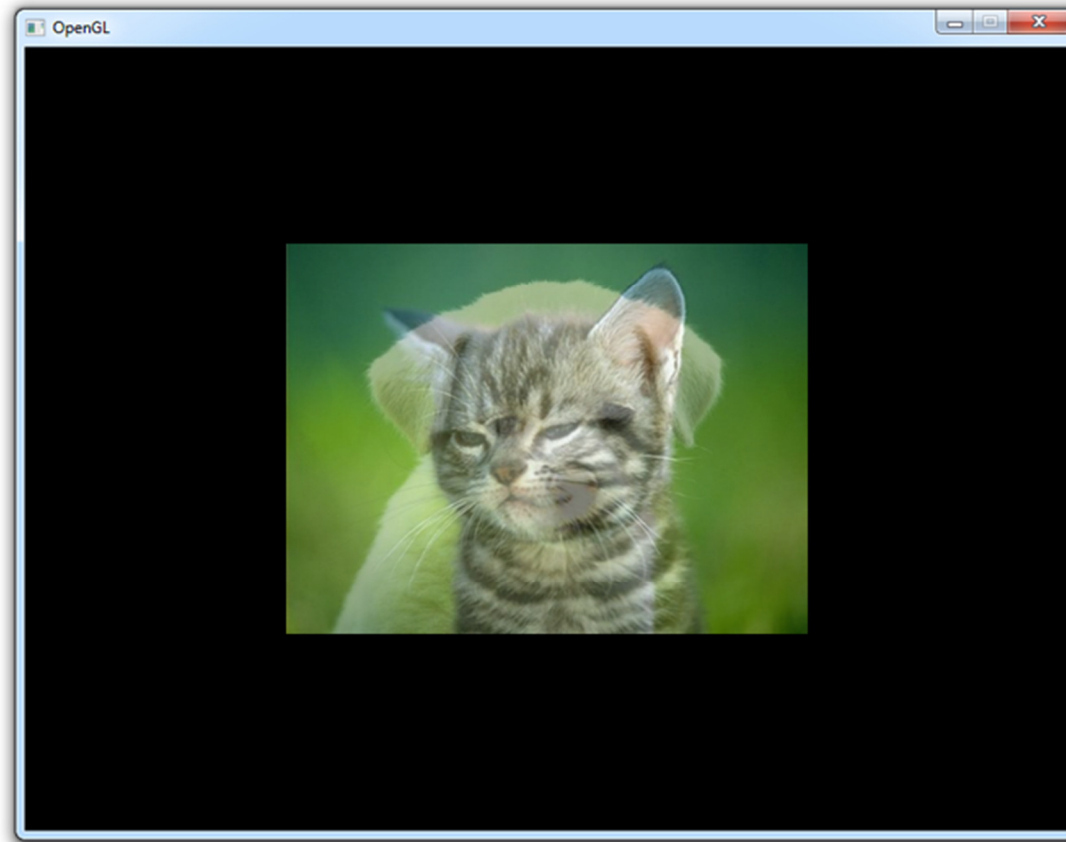
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("sample.png", &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texKitten"), 0);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("sample2.png", &width, &height, 0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texPuppy"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

The texture units of the samplers are set using the `glUniform` function you've seen in the previous chapter. It simply accepts an integer specifying the texture unit. Make sure that at least the wrap texture parameters are set for both textures. This code should result in the following image.



As always, have a look at the sample [source code](#) if you have trouble getting the program to work.

Now that texture sampling has been covered in this chapter, you're finally ready to dive into transformations and ultimately 3D. The knowledge you have at this point should be sufficient for producing most types of 2D games, except for transformations like rotation and scaling which will be covered in the [next chapter](#).

Exercises

- Animate the blending between the textures by adding a `time` uniform. ([Solution](#))
- Draw a reflection of the kitten in the lower half of the rectangle. ([Solution](#))
- Now try adding distortion with `sin` and the time variable to simulate water. ([Expected result](#), [Solution](#))

Matrices

Since this is a guide on graphics programming, this chapter will not cover a lot of the extensive theory behind matrices. Only the theory that applies to their use in computer graphics will be considered here and they will be explained from a programmer's perspective. If you want to learn more about the topic, [these Khan Academy videos](#) are a really good general introduction to the subject.

A matrix is a rectangular array of mathematical expressions, much like a two-dimensional array. Below is an example of a matrix displayed in the common square brackets form.

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Matrices values are indexed by (i, j) where i is the row and j is the column. That is why the matrix displayed above is called a 3-by-2 matrix. To refer to a specific value in the matrix, for example 5, the a_{31} (a_{31}) notation is used.

Basic operations

To get a bit more familiar with the concept of an array of numbers, let's first look at a few basic operations.

Addition and subtraction

Just like regular numbers, the addition and subtraction operators are also defined for matrices. The only requirement is that the two operands have exactly the same row and column dimensions.

$$\begin{bmatrix} 3 & 2 \\ 0 & 4 \end{bmatrix} + \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 3+4 & 2+2 \\ 0+2 & 4+2 \end{bmatrix} = \begin{bmatrix} 7 & 4 \\ 2 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 2 \\ 2 & 7 \end{bmatrix} - \begin{bmatrix} 3 & 2 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} 4-3 & 2-2 \\ 2-0 & 7-4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}$$

The values in the matrices are individually added or subtracted from each other.

Scalar product

The product of a scalar and a matrix is as straightforward as addition and subtraction.

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

The values in the matrices are each multiplied by the scalar.

Matrix-Vector product

The product of a matrix with another matrix is quite a bit more involved and is often misunderstood, so for simplicity's sake I will only mention the specific cases that apply to graphics programming.

To see how matrices are actually used to transform vectors, we'll first dive into the product of a matrix and a vector.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} a \cdot x + b \cdot y + c \cdot z + d \cdot 1 \\ e \cdot x + f \cdot y + g \cdot z + h \cdot 1 \\ i \cdot x + j \cdot y + k \cdot z + l \cdot 1 \\ m \cdot x + n \cdot y + o \cdot z + p \cdot 1 \end{pmatrix}$$

To calculate the product of a matrix and a vector, the vector is written as a 4-by-1 matrix.

The expressions to the right of the equals sign show how the new \boxed{x} , \boxed{y} and \boxed{z} values are calculated after the vector has been transformed.

For those among you who aren't very math savvy, the dot is a multiplication sign.

I will mention each of the common vector transformations in this section and how a matrix can be formed that performs them.

For completeness, let's first consider a transformation that does absolutely nothing.

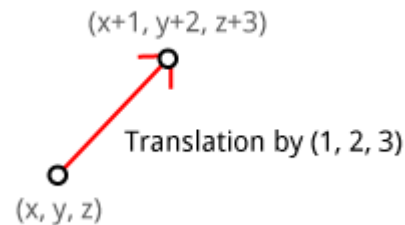
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot x + 0 \cdot y + 0 \cdot z + 0 \cdot 1 \\ 0 \cdot x + 1 \cdot y + 0 \cdot z + 0 \cdot 1 \\ 0 \cdot x + 0 \cdot y + 1 \cdot z + 0 \cdot 1 \\ 0 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot x \\ 1 \cdot y \\ 1 \cdot z \\ 1 \cdot 1 \end{pmatrix}$$

This matrix is called the **identity matrix**, because just like the number **1**, it will always return the value it was originally multiplied by.

Let's look at the most common vector transformations now and deduce how a matrix can be formed from them.

Translation

To see why we're working with 4-by-1 vectors and subsequently 4-by-4 transformation matrices, let's see how a translation matrix is formed. A **translation moves** a vector a certain **distance** in a certain **direction**.



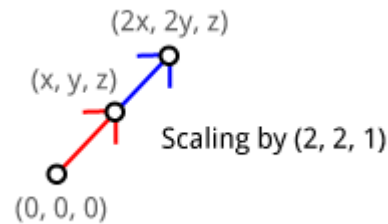
Can you guess from the multiplication overview what the matrix should look like to translate a vector by (x, y, z) ?

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + X \cdot 1 \\ y + Y \cdot 1 \\ z + Z \cdot 1 \\ 1 \end{pmatrix}$$

Without the fourth column and the bottom **1** value a translation wouldn't have been possible.

Scaling

A **scale transformation** scales each of a vector's components by a (different) scalar. It is commonly used to **shrink or stretch** a vector as demonstrated below.



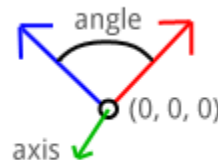
If you understand how the previous matrix was formed, it should not be difficult to come up with a matrix that scales a given vector by (SX, SY, SZ) .

$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} SX \cdot x \\ SY \cdot y \\ SZ \cdot z \\ 1 \end{pmatrix}$$

If you think about it for a moment, you can see that scaling would also be possible with a mere 3-by-3 matrix.

Rotation

A **rotation transformation** rotates a vector around the **origin** $(0, 0, 0)$ using a given **axis and angle**. To understand how the axis and the angle control a rotation, let's do a small experiment.



Put your thumb up against your monitor and try rotating your hand around it. The object, your hand, is rotating around your thumb: the rotation axis. The further you rotate your hand away from its initial position, the higher the rotation angle.

In this way the rotation axis can be imagined as an arrow an object is rotating around. **If you imagine your monitor to be a 2-dimensional XY surface, the rotation axis (your thumb) is pointing in the Z direction.**

Objects can be rotated around any given axis, but for now only the X, Y and Z axis are important. You'll see later in this chapter that any rotation axis can be established by rotating around the X, Y and Z axis simultaneously.

The matrices for rotating around the three axes are specified here. The rotation angle is indicated by the theta (θ).

Rotation around X-axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around Y-axis:

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x + \sin \theta \cdot z \\ y \\ -\sin \theta \cdot x + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around Z-axis:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x - \sin \theta \cdot y \\ \sin \theta \cdot x + \cos \theta \cdot y \\ z \\ 1 \end{pmatrix}$$

Don't worry about understanding the actual geometry behind this, explaining that is beyond the scope of this guide. What matters is that you have a solid idea of how a rotation is described by a rotation axis and an angle and that you've at least seen what a rotation matrix looks like.

Matrix-Matrix product

In the previous section you've seen how transformation matrices can be used to apply transformations to vectors, but this by itself is not very useful. It clearly takes far less effort to do a translation and scaling by hand without all those pesky matrices!

Now, what if I told you that **it is possible to combine as many transformations as you want into a single matrix by simply multiplying them**? You would be able to apply even the most complex transformations to any vertex with a simple multiplication.

In the same style as the previous section, this is how the product of two 4-by-4 matrices is determined:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} = \begin{bmatrix} aA + bE + cI + dM & aB + bF + cJ + dN & aC + bG + cK + dO & aD + bH + cL + dP \\ eA + fE + gI + hM & eB + fF + gJ + hN & eC + fG + gK + hO & eD + fH + gL + hP \\ iA + jE + kI + lM & iB + jF + kJ + lN & iC + jG + kK + lO & iD + jH + kL + lP \\ mA + nE + oI + pM & mB + nF + oJ + pN & mC + nG + oK + pO & mD + nH + oL + pP \end{bmatrix}$$

The above is commonly recognized among mathematicians as an *indecipherable mess*. To get a better idea of what's going on, let's consider two 2-by-2 matrices instead.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 \cdot a + 2 \cdot c & 1 \cdot b + 2 \cdot d \\ 3 \cdot a + 4 \cdot c & 3 \cdot b + 4 \cdot d \end{bmatrix}$$

Try to see the pattern here with help of the colors. The factors on the left side (1, 2 and 3, 4) of the multiplication dot are the values in the row of the first matrix. The factors on the right side are the values in the rows of the second matrix repeatedly. It is not necessary to remember how exactly this works, but it's good to have seen how it's done at least once.

Combining transformations

To demonstrate the multiplication of two matrices, let's try scaling a given vector by (2, 2, 2) and translating it by (1, 2, 3). Given the translation and scaling matrices above, the following product is calculated:

$$M_{\text{translate}} \cdot M_{\text{scale}} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice how we want to scale the vector first, but the scale transformation comes last in the multiplication. Pay attention to this when combining transformations or you'll get the opposite of what you've asked for.

Now, let's try to transform a vector and see if it worked:

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x + 1 \\ 2y + 2 \\ 2z + 3 \\ 1 \end{pmatrix}$$

Perfect! The vector is first scaled by two and then shifted in position by (1, 2, 3).

Transformations in OpenGL

You've seen in the previous sections how basic transformations can be applied to vectors to move them around in the world.

The job of transforming 3D points into 2D coordinates on your screen is also accomplished through matrix transformations. Just like the graphics pipeline, transforming a vector is done step-by-step. Although OpenGL allows you to decide on these steps yourself, all 3D graphics applications use a variation of the process described here.



Each transformation transforms a vector into a new coordinate system, thus moving to the next step. These transformations and coordinate systems will be discussed below in more detail.

Model matrix

The model matrix transforms a position in a model to the position in the world. This position is affected by the position, scale and rotation of the model that is being drawn. It is generally a combination of the simple transformations you've seen before. If you are already specifying your vertices in world coordinates (common when drawing a simple test scene), then this matrix can simply be set to the identity matrix.

View matrix

In real life you're used to moving the camera to alter the view of a certain scene, in OpenGL it's the other way around. **The camera in OpenGL cannot move and is defined to be located at $(0, 0, 0)$ facing the negative Z direction.** That means that instead of moving and rotating the camera, the world is moved and rotated around the camera to construct the appropriate view.

Older versions of OpenGL forced you to use **ModelView** and **Projection** transformations. The ModelView matrix combined the model and view transformations into one. **I personally find it is easier to separate the two, so the view transformation can be modified independently of the model matrix.**

That means that to simulate a camera transformation, you actually have to transform the world with the inverse of that transformation. Example: if you want to move the camera up, you have to move the world down instead.

Projection matrix

After the world has been aligned with your camera using the view transformation, the projection transformation can be applied, resulting in the clip coordinates. If you're doing a perspective transformation, these clip coordinates are not ready to be used as normalized device coordinates just yet.

To transform the clipping coordinate into a normalized device coordinate, *perspective division* has to be performed.

- A clipping coordinate resulting from a perspective projection has a number different than 1 in the fourth row, also known as **w**.
- This number directly reflects the effect of objects further away being smaller than those up front.

$$v_{\text{normalized}} = \begin{pmatrix} x_{\text{clip}}/w_{\text{clip}} \\ y_{\text{clip}}/w_{\text{clip}} \\ z_{\text{clip}}/w_{\text{clip}} \end{pmatrix}$$

The **x** and **y** coordinates will be in the familiar **-1** and **1** range now, which OpenGL can transform into window coordinates. The **z** is known as the depth and will play an important role in the next chapter.

The coordinates resulting from the projection transformation are called clipping coordinates because the value of **w** is used to determine whether an object is too close or behind the camera or too far away to be drawn. The projection matrix is created with those limits, so you'll be able to specify these yourself.

Putting it all together

To sum it all up, the final transformation of a vertex is the product of the model, view and projection matrices.

$$v' = M_{\text{proj}} \cdot M_{\text{view}} \cdot M_{\text{model}} \cdot v$$

This operation is typically performed in the vertex shader and assigned to the **gl_Position** return value in clipping coordinates.

OpenGL will perform the perspective division and transformation into window coordinates. It is important to be aware of these steps, because you'll have to do them yourself when working with techniques like shadow mapping.

Using transformations for 3D

Now that you know three important transformations, it is time to implement these in code to create an actual 3D scene. You can use any of the programs developed in the last two chapters as a base, but I'll use the texture blending sample from the end of the last chapter here.

To introduce matrices in the code, we can make use of the GLM (OpenGL Math) library.

- This library comes with **vector and matrix classes** and will handle all the math efficiently without ever having to worry about it.
- It is a **header-only library**, which means you don't have to link with anything.

To use it, add the GLM root directory to your include path and include these three headers:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

- The second header **includes functions to ease the calculation of the view and projection matrices.**
- The third header **adds functionality for converting a matrix object into a float array for usage in OpenGL.**

A simple transformation

Before diving straight into 3D, let's first try a **simple 2D rotation.**

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(180.0f), glm::vec3(0.0f, 0.0f, 1.0f));
```

The first line creates a new 4-by-4 matrix and initializes it to the identity matrix.

The **glm::rotate** function multiplies this matrix by a rotation transformation of 180 degrees around the Z axis. Remember that since the screen lies in the XY plane, the Z axis is the axis you want to rotate points around.

To see if it works, let's try to rotate a vector with this transformation:

```
glm::vec4 result = trans * glm::vec4(1.0f, 0.0f, 0.0f, 1.0f);  
printf("%f, %f, %f\n", result.x, result.y, result.z);
```

As expected, the output is `(-1, 0, 0)`.

- A counter-clockwise rotation of 180 degrees of a vector pointing to the right results in a vector pointing to the left.
- Note that the rotation would be clockwise if an axis `(0, 0, -1)` was used.

The next step is to perform this transformation in the vertex shader to rotate every drawn vertex.

- GLSL has a special `mat4` type to hold matrices and we can use that to upload the transformation to the GPU as uniform.

```
GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");  
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(trans));
```

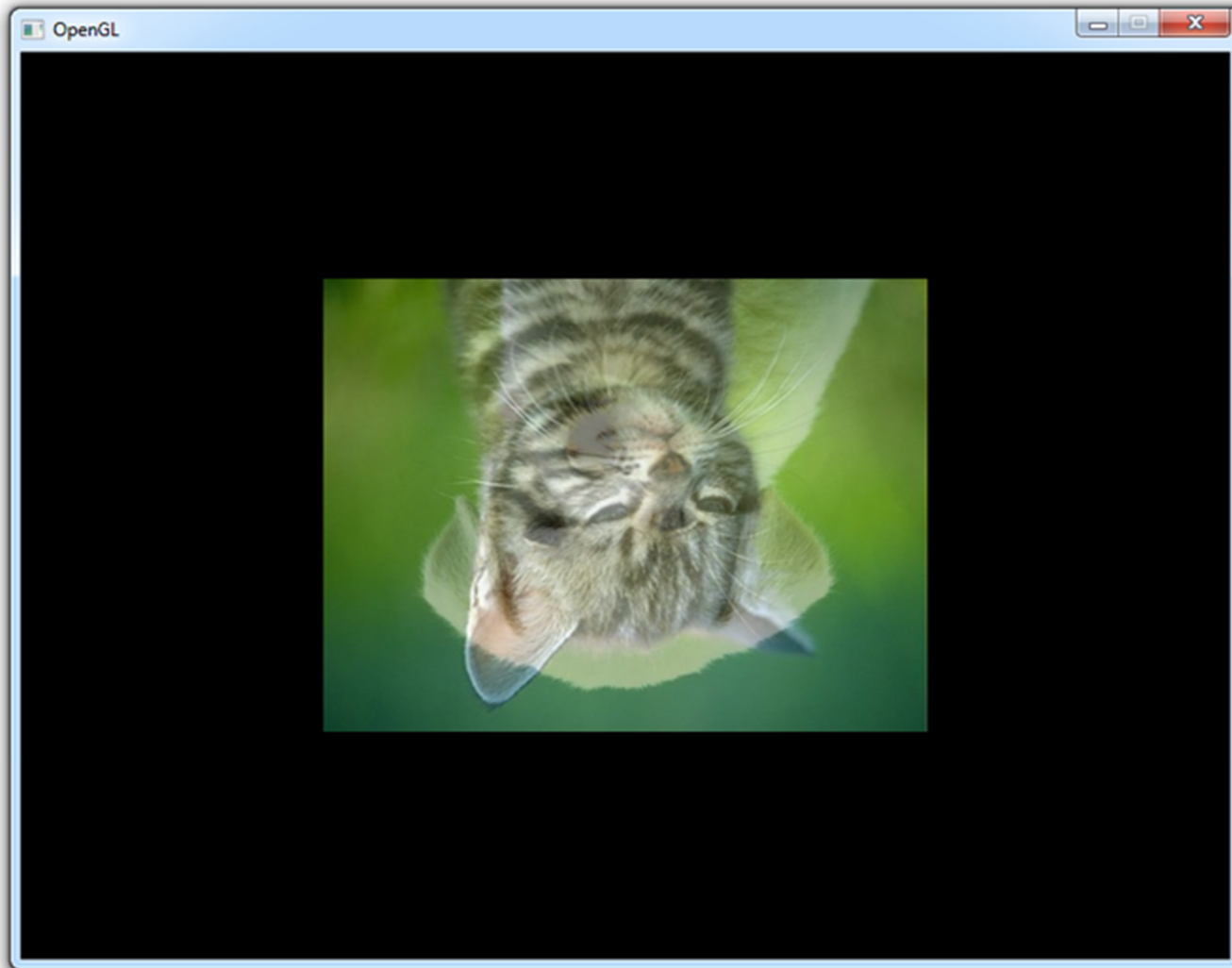
- The second parameter of the `glUniformMatrix4fv` function specifies how many matrices are to be uploaded, because you can have arrays of matrices in GLSL.
- The third parameter specifies whether the specified matrix should be transposed before usage. This is related to the way matrices are stored as `float` arrays in memory; you don't have to worry about it.
- The last parameter specifies the matrix to upload, where the `glm::value_ptr` function converts the matrix class into an array of 16 (4x4) floats.

All that remains is updating the vertex shader to include this uniform and use it to transform each vertex:

```
#version 150 core  
  
in vec2 position;  
in vec3 color;  
in vec2 texcoord;  
out vec3 Color;  
out vec2 Texcoord;  
  
uniform mat4 trans;
```

```
void main()
{
    Color = color;
    Texcoord = texcoord;
    gl_Position = trans * vec4(position, 0.0, 1.0);
}
```

The primitives in your scene will now be upside down.

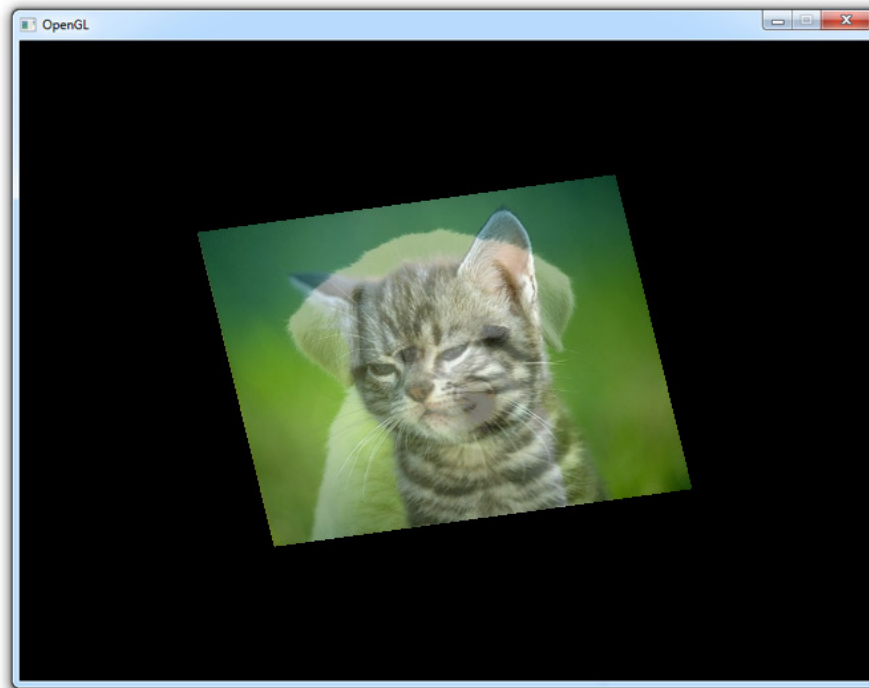


To spice things up a bit, you could change the rotation with time:

```
auto t_start = std::chrono::high_resolution_clock::now();  
...  
// Calculate transformation  
auto t_now = std::chrono::high_resolution_clock::now();  
float time = std::chrono::duration_cast<std::chrono::duration<float>>(t_now - t_start).count();
```

```
glm::mat4 trans;  
trans = glm::rotate(  
    trans,  
    time * glm::radians(180.0f),  
    glm::vec3(0.0f, 0.0f, 1.0f)  
);  
  
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(trans));  
  
// Draw a rectangle from the 2 triangles using 6 indices  
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);  
  
...
```

This will result in something like this:



This will result in something like this:

You can find the full code [here](#) if you have any issues.

Going 3D

The rotation above can be considered the model transformation, because it transforms the vertices in object space to world space using the rotation of the object

You can find the full code [here](#) if you have any issues.

```
glm::mat4 view = glm::lookAt(
    glm::vec3(1.2f, 1.2f, 1.2f),
    glm::vec3(0.0f, 0.0f, 0.0f),
    glm::vec3(0.0f, 0.0f, 1.0f)
);
GLint uniView = glGetUniformLocation(shaderProgram, "view");
glUniformMatrix4fv(uniView, 1, GL_FALSE, glm::value_ptr(view));
```

To create the view transformation, GLM offers the useful `glm::lookAt` function that simulates a moving camera.

- The first parameter specifies the position of the camera,
- the second the point to be centered on-screen
- and the third the up axis. Here up is defined as the Z axis, which implies that the XY plane is the "ground".

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 1.0f, 10.0f);
GLint uniProj = glGetUniformLocation(shaderProgram, "proj");
glUniformMatrix4fv(uniProj, 1, GL_FALSE, glm::value_ptr(proj));
```

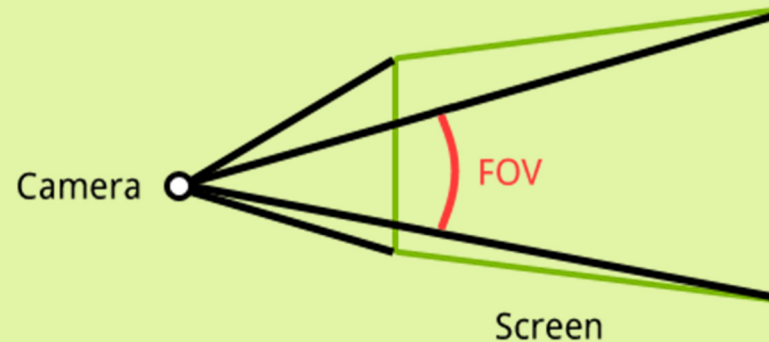
Similarly, GLM comes with the `glm::perspective` function to create a perspective projection matrix.

- The first parameter is the vertical field-of-view,
- the second parameter the aspect ratio of the screen
- and the last two parameters are the near and far planes.

Field-of-view

The field-of-view defines the angle between the top and bottom of the 2D surface on which the world will be projected.

- Zooming in games is often accomplished by decreasing this angle as opposed to moving the camera closer, because it more closely resembles real life.



- By decreasing the angle, you can imagine that the "rays" from the camera spread out less and thus cover a smaller area of the scene.
- The near and far planes are known as the clipping planes. Any vertex closer to the camera than the **near** clipping plane and any vertex farther away than the **far** clipping plane is clipped as these influence the **w** value.

Now piecing it all together, the vertex shader looks something like this:

```
#version 150 core

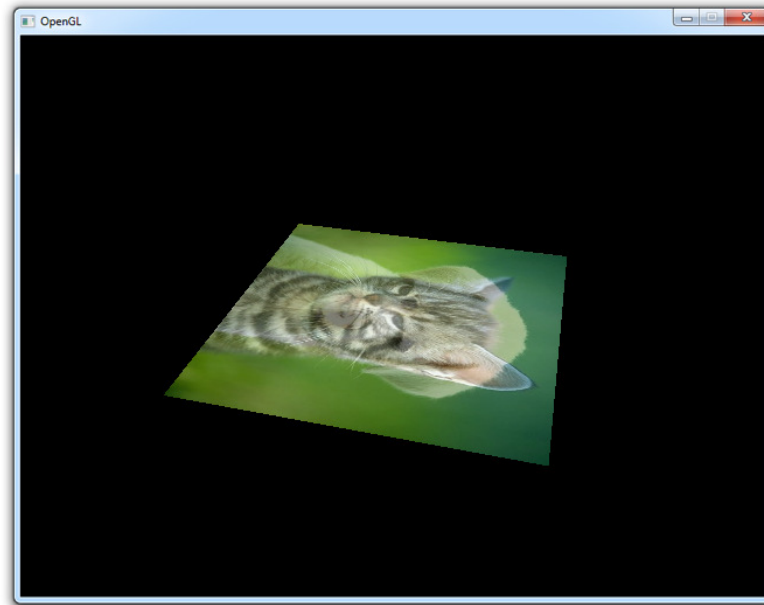
in vec2 position;
in vec3 color;
in vec2 texcoord;

out vec3 Color;
out vec2 Texcoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 proj;
```

```
void main()
{
    Color = color;
    Texcoord = texcoord;
    gl_Position = proj * view * model * vec4(position, 0.0, 1.0);
}
```

Notice that I've renamed the matrix previously known as `trans` to `model` and it is still updated every frame.



Success! You can find the full code [here](#) if you get stuck.

Exercises

- Make the rectangle with the blended image grow bigger and smaller with `sin`. ([Solution](#))
- Make the rectangle flip around the X axis after pressing the space bar and slowly stop again. ([Solution](#))

Extra buffers

Up until now there is only one type of output buffer you've made use of, the color buffer.

This chapter will discuss two additional types, the *depth buffer* and the *stencil buffer*. For each of these a problem will be presented and subsequently solved with that specific buffer.

Preparations

To best demonstrate the use of these buffers, let's draw a cube instead of a flat shape.

- The vertex shader needs to be modified to accept a third coordinate:

```
in vec3 position;  
...  
gl_Position = proj * view * model * vec4(position, 1.0);
```

We're also going to need to alter the color again later in this chapter, so make sure the fragment shader multiplies the texture color by the color attribute:

```
vec4 texColor = mix(texture(texKitten, Texcoord), texture(texPuppy, Texcoord), 0.5);  
outColor = vec4(Color, 1.0) * texColor;
```

Vertices are now 8 floats in size, so you'll have to update the vertex attribute offsets and strides as well. Finally, add the extra coordinate to the vertex array:

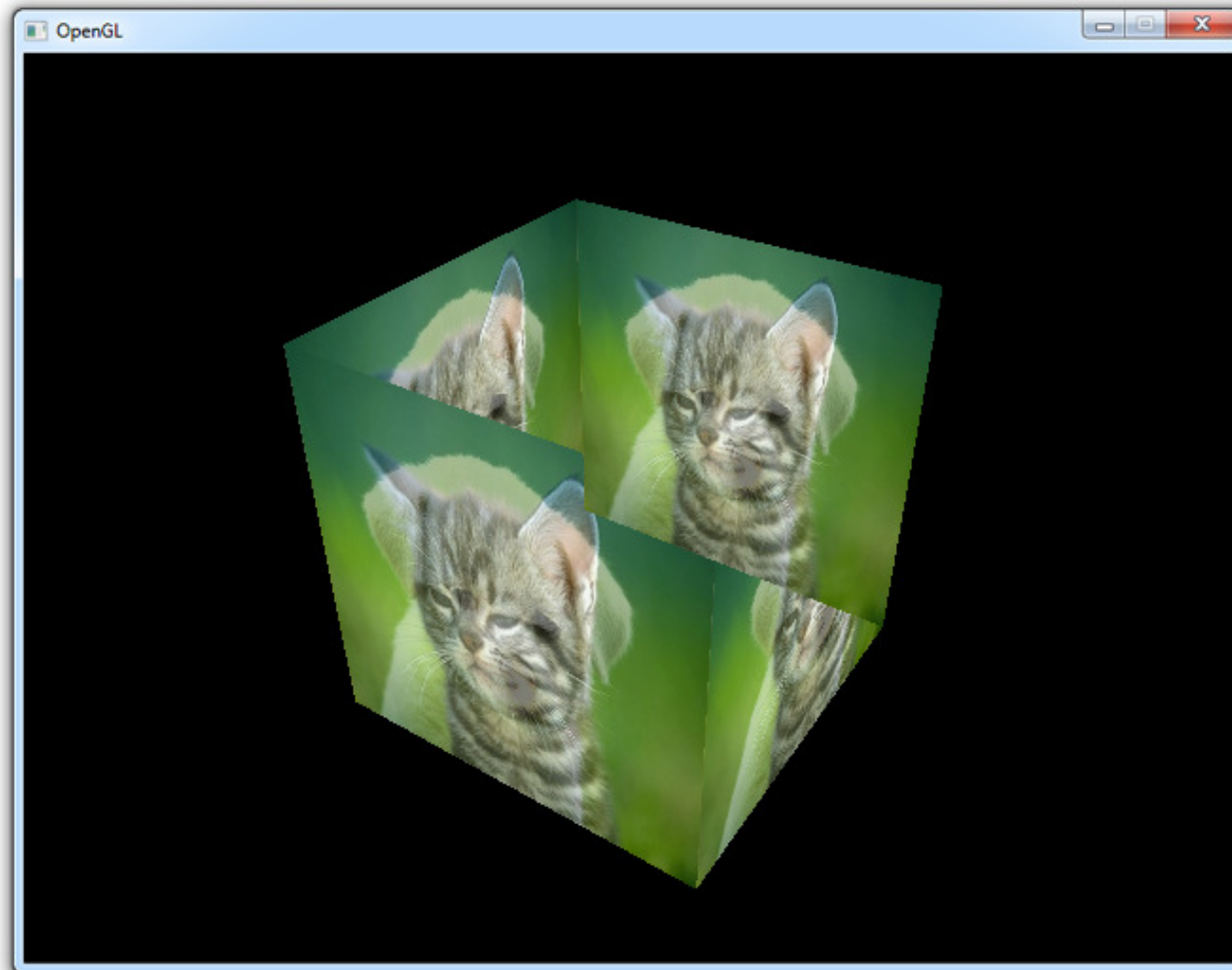
```
float vertices[] = {  
    // X   Y   Z   R   G   B   U   V  
    -0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,  
     0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f,  
     0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f,  
    -0.5f, -0.5f, 0.0f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f  
};
```


Confirm that you've made all the required changes by running your program and checking if it still draws a flat spinning image of a kitten blended with a puppy.

A single cube consists of 36 vertices (6 sides * 2 triangles * 3 vertices), so I will ease your life by providing the array [here](#).

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

We will not make use of element buffers for drawing this cube, so you can use `glDrawArrays` to draw it. If you were confused by this explanation, you can compare your program to [this reference code](#).



It immediately becomes clear that the cube is not rendered as expected when seeing the output.

- The sides of the cube are being drawn, but they overlap each other in strange ways!
- The problem here is that when OpenGL draws your cube triangle-by-triangle, it will simply write over pixels even though something else may have been drawn there before.
- In this case OpenGL will happily draw triangles in the back over triangles at the front.

Luckily OpenGL offers ways of telling it when to draw over a pixel and when not to. I'll go over the two most important ways of doing that, depth testing and stencil, in this chapter.

Depth buffer

Z-buffering is a way of keeping track of the depth of every pixel on the screen.

- The depth is an increasing function of the distance between the screen plane and a fragment that has been drawn.
- That means that the fragments on the sides of the cube further away from the viewer have a higher depth value, whereas fragments closer have a lower depth value.

If this depth is stored along with the color when a fragment is written, fragments drawn later can compare their depth to the existing depth to determine if the new fragment is closer to the viewer than the old fragment.

- If that is the case, it should be drawn over and otherwise it can simply be discarded. This is known as **depth testing**.

OpenGL offers a way to store these depth values in an extra buffer, called the **depth buffer**, and perform the required check for fragments automatically.

- The **fragment shader** will not run for fragments that are invisible, which can have a significant impact on performance. This functionality can be enabled by calling **glEnable**.

```
glEnable(GL_DEPTH_TEST);
```

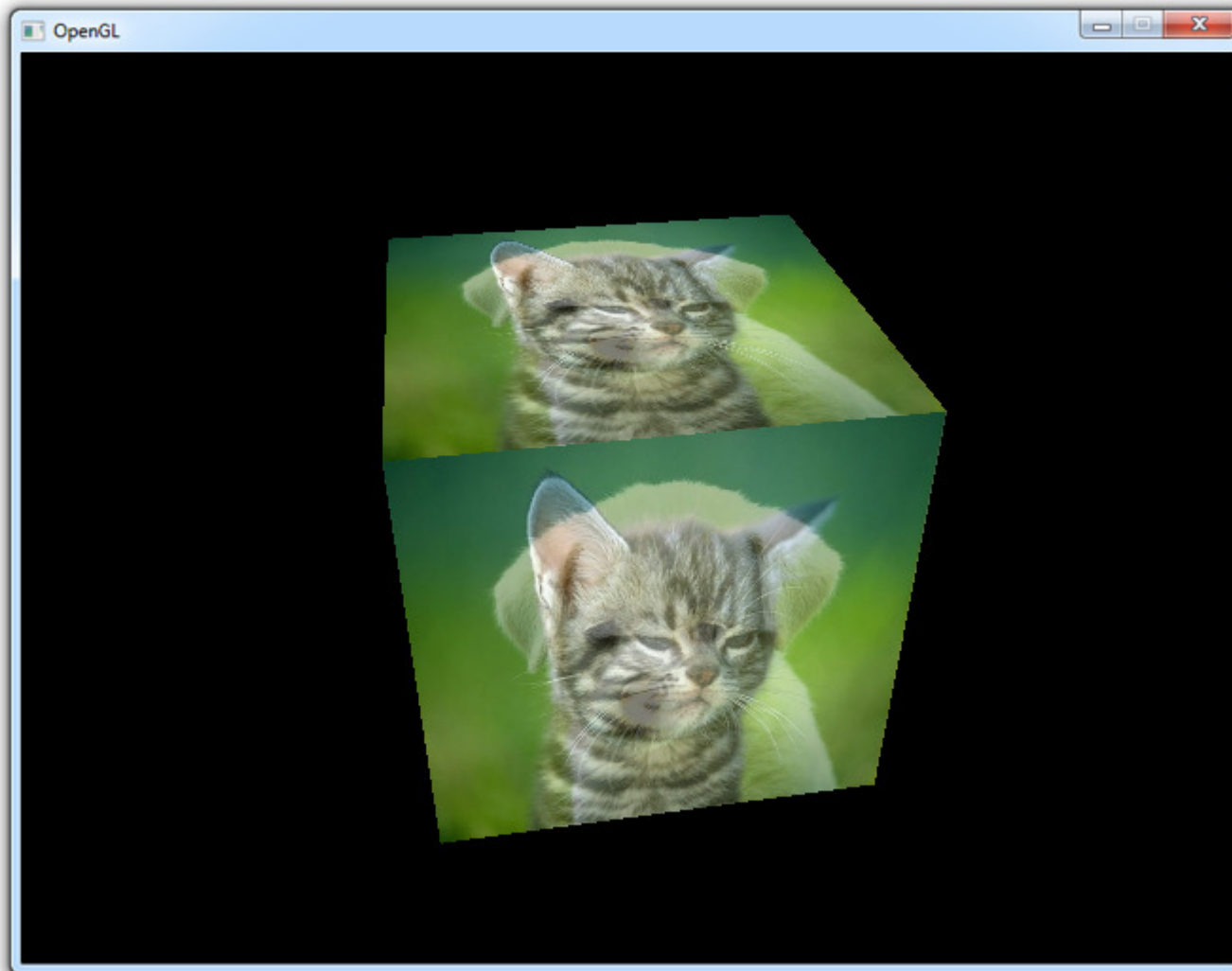
If you enable this functionality now and run your application, you'll notice that you get a black screen.

- That happens because the depth buffer is filled with 0 depth for each pixel by default.
- Since no fragments will ever be closer than that they are all discarded.

- The depth buffer can be cleared along with the color buffer by extending the `glClear` call:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

The default clear value for the depth is `1.0f`, which is equal to the depth of your far clipping plane and thus the furthest depth that can be represented. All fragments will be closer than that, so they will no longer be discarded.



With the depth test capability enabled, the cube is now rendered correctly.

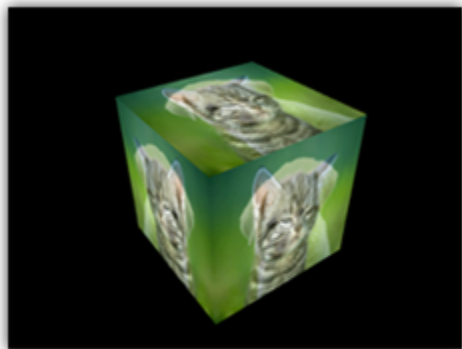
- Just like the color buffer, the depth buffer has a certain amount of bits of precision which can be specified by you.
- Less bits of precision reduce the extra memory use, but can introduce rendering errors in more complex scenes.

Stencil buffer

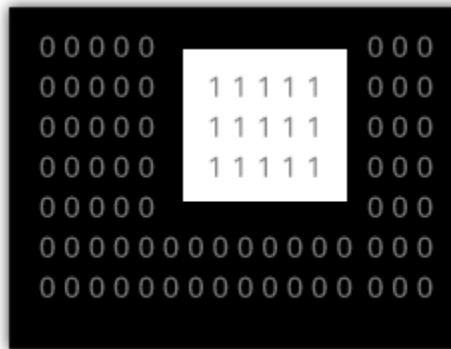
The stencil buffer is an optional extension of the depth buffer that gives you more control over the question of which fragments should be drawn and which shouldn't.

- Like the depth buffer, a value is stored for every pixel, but this time you get to control when and how this value changes and when a fragment should be drawn depending on this value.
- Note that if the depth test fails, the stencil test no longer determines whether a fragment is drawn or not, but these fragments can still affect values in the stencil buffer!

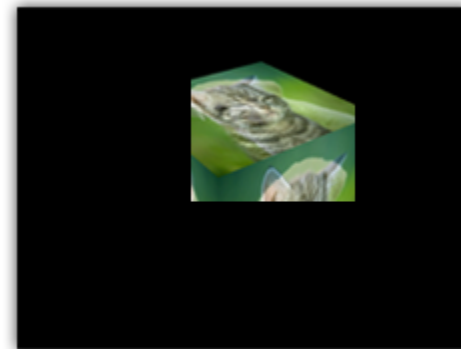
To get a bit more acquainted with the stencil buffer before using it, let's start by analyzing a simple example.



Color buffer without stencil test



Stencil buffer



Color buffer with stencil test

In this case the stencil buffer was first cleared with zeroes and then a rectangle of ones was drawn to it.

- The drawing operation of the cube uses the values from the stencil buffer to only draw fragments with a stencil value of 1.
- Now that you have an understanding of what the stencil buffer does, we'll look at the relevant OpenGL calls.

```
glEnable(GL_STENCIL_TEST);
```

Stencil testing is enabled with a call to `glEnable`, just like depth testing.

- You don't have to add this call to your code just yet. I'll first go over the API details in the next two sections and then we'll make a cool demo.

Setting values

Regular drawing operations are used to determine which values in the stencil buffer are affected by any stencil operation.

- If you want to affect a rectangle of values like in the sample above, simply draw a 2D quad in that area.
- What happens to those values can be controlled by you using the `glStencilFunc`, `glStencilOp` and `glStencilMask` functions.

The `glStencilFunc` call is used to specify the conditions under which a fragment passes the stencil test. Its parameters are discussed below.

- `func`: The test function, can be `GL_NEVER`, `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, and `GL_ALWAYS`.
- `ref`: A value to compare the stencil value to using the test function.
- `mask`: A bitwise AND operation is performed on the stencil value and reference value with this mask value before comparing them.
- If you don't want stencils with a value lower than 2 to be affected, you would use:

```
glStencilFunc(GL_GEQUAL, 2, 0xFF);
```

The mask value is set to all ones (in case of an 8-bit stencil buffer), so it will not affect the test.

The `glStencilOp` call specifies what should happen to stencil values depending on the outcome of the stencil and depth tests. The parameters are:

- `sfail`: Action to take if the stencil test fails.
- `dpfail`: Action to take if the stencil test is successful, but the depth test failed.
- `dppass`: Action to take if both the stencil test and depth tests pass.

Stencil values can be modified in the following ways:

- `GL_KEEP` : The current value is kept.
- `GL_ZERO` : The stencil value is set to 0.
- `GL_REPLACE` : The stencil value is set to the reference value in the `glStencilFunc` call.
- `GL_INCR` : The stencil value is increased by 1 if it is lower than the maximum value.
- `GL_INCR_WRAP` : Same as `GL_INCR`, with the exception that the value is set to 0 if the maximum value is exceeded.
- `GL_DECR` : The stencil value is decreased by 1 if it is higher than 0.
- `GL_DECR_WRAP` : Same as `GL_DECR`, with the exception that the value is set to the maximum value if the current value is 0 (the stencil buffer stores unsigned integers).
- `GL_INVERT` : A bitwise invert is applied to the value.

Finally, `glStencilMask` can be used to control the bits that are written to the stencil buffer when an operation is run. The default value is all ones, which means that the outcome of any operation is unaffected.

If, like in the example, you want to set all stencil values in a rectangular area to 1, you would use the following calls

```
glStencilFunc(GL_ALWAYS, 1, 0xFF);  
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);  
glStencilMask(0xFF);
```

In this case the rectangle shouldn't actually be drawn to the color buffer, since it is only used to determine which stencil values should be affected.

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);  
glDepthMask(GL_FALSE);
```

The `glColorMask` function allows you to specify which data is written to the color buffer during a drawing operation. In this case you would want to disable all color channels (red, green, blue, alpha). Writing to the depth buffer needs to be disabled separately as well with `glDepthMask`, so that cube drawing operation won't be affected by leftover depth values of the rectangle. This is cleaner than simply clearing the depth buffer again later.

Using values in drawing operations

With the knowledge about setting values, using them for testing fragments in drawing operations becomes very simple. All you need to do now is re-enable color and depth writing if you had disabled those earlier and setting the test function to determine which fragments are drawn based on the values in the stencil buffer.

```
glStencilFunc(GL_EQUAL, 1, 0xFF);
```

If you use this call to set the test function, the stencil test will only pass for pixels with a stencil value equal to 1. A fragment will only be drawn if it passes both the stencil and depth test, so setting the `glStencilOp` is not necessary. In the case of the example above only the stencil values in the rectangular area were set to 1, so only the cube fragments in that area will be drawn.

```
glStencilMask(0x00);
```

One small detail that is easy to overlook is that the cube draw call could still affect values in the stencil buffer. This problem can be solved by setting the stencil bit mask to all zeroes, which effectively disables stencil writing.

Planar reflections

Let's spice up the demo we have right now a bit by adding a floor with a reflection under the cube. I'll add the vertices for the floor to the same vertex buffer the cube is currently using to keep things simple:

```
float vertices[] = {  
    ...  
  
    -1.0f, -1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,  
    1.0f, -1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f,  
    1.0f, 1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,  
    1.0f, 1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,  
    -1.0f, 1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,  
    -1.0f, -1.0f, -0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f  
}
```

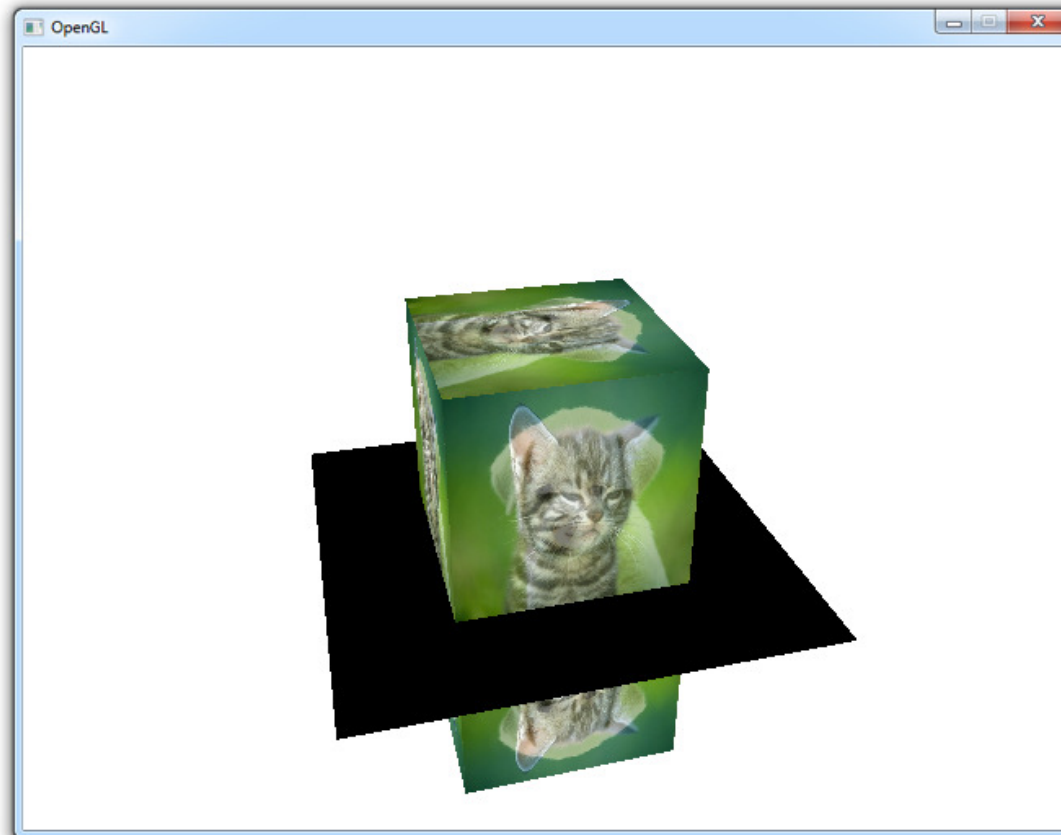
Now add the extra draw call to your main loop:

```
glDrawArrays(GL_TRIANGLES, 36, 6);
```

To create the reflection of the cube itself, it is sufficient to draw it again but inverted on the Z-axis:

```
model = glm::scale(  
    glm::translate(model, glm::vec3(0, 0, -1)),  
    glm::vec3(1, 1, -1)  
);  
glUniformMatrix4fv(uniModel, 1, GL_FALSE, glm::value_ptr(model));  
glDrawArrays(GL_TRIANGLES, 0, 36);
```

I've set the color of the floor vertices to black so that the floor does not display the texture image, so you'll want to change the clear color to white to be able to see it. I've also changed the camera parameters a bit to get a good view of the scene.



Two issues are noticeable in the rendered image:

- The floor occludes the reflection because of depth testing.
- The reflection is visible outside of the floor.

The first problem is easy to solve by temporarily disabling writing to the depth buffer when drawing the floor:

```
glDepthMask(GL_FALSE);  
glDrawArrays(GL_TRIANGLES, 36, 6);  
glDepthMask(GL_TRUE);
```

To fix the second problem, it is necessary to discard fragments that fall outside of the floor. Sounds like it's time to see what stencil testing is really worth!

It can be greatly beneficial at times like these to make a little list of the rendering stages of the scene to get a proper idea of what is going on.

- Draw regular cube.
- Enable stencil testing and set test function and operations to write ones to all selected stencils.
- Draw floor.
- Set stencil function to pass if stencil value equals 1.
- Draw inverted cube.
- Disable stencil testing.

The new drawing code looks like this:

```
glEnable(GL_STENCIL_TEST);  
  
// Draw floor  
glStencilFunc(GL_ALWAYS, 1, 0xFF); // Set any stencil to 1  
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);  
glStencilMask(0xFF); // Write to stencil buffer  
glDepthMask(GL_FALSE); // Don't write to depth buffer  
glClear(GL_STENCIL_BUFFER_BIT); // Clear stencil buffer (0 by default)  
  
glDrawArrays(GL_TRIANGLES, 36, 6);
```

```
// Draw cube reflection
glStencilFunc(GL_EQUAL, 1, 0xFF); // Pass test if stencil value is 1
glStencilMask(0x00); // Don't write anything to stencil buffer
glDepthMask(GL_TRUE); // Write to depth buffer

model = glm::scale(
    glm::translate(model, glm::vec3(0, 0, -1)),
    glm::vec3(1, 1, -1)
);
glUniformMatrix4fv(uniModel, 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

glDisable(GL_STENCIL_TEST);
```

I've annotated the code above with comments, but the steps should be mostly clear from the stencil buffer section.

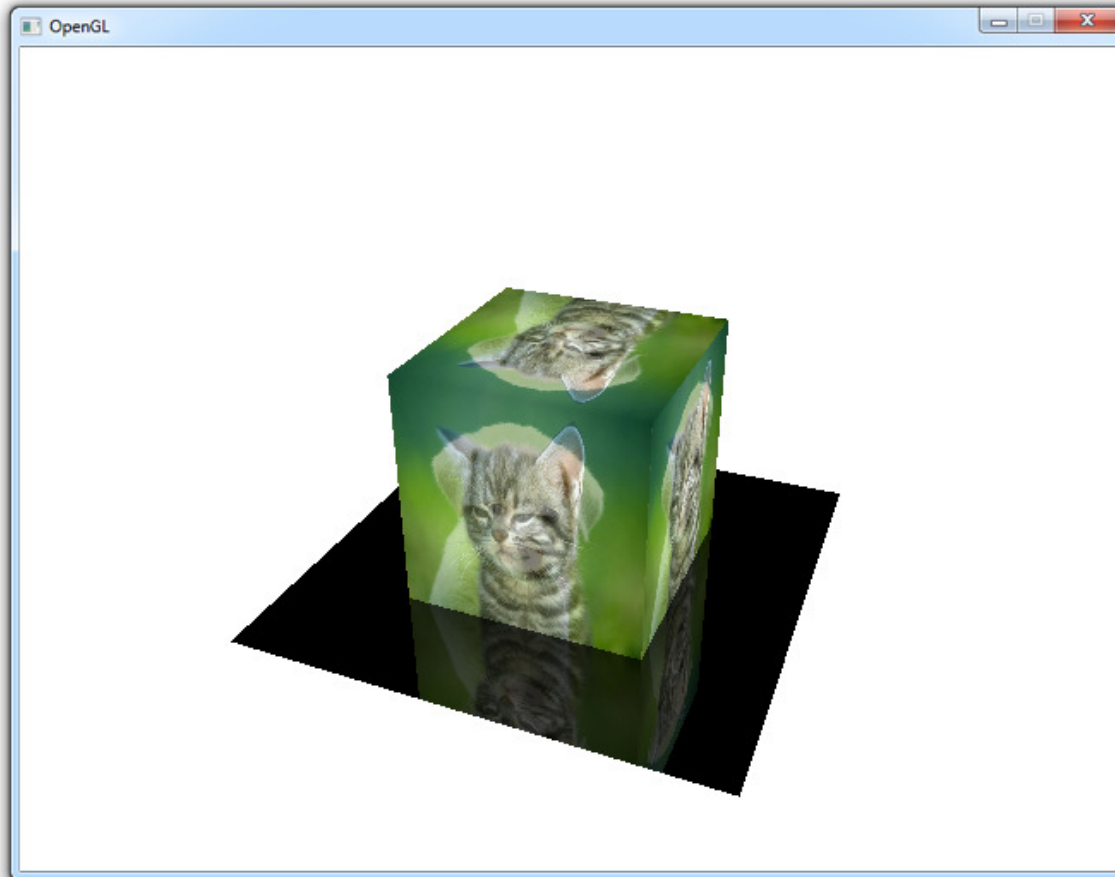
Now just one final touch is required, to darken the reflected cube a little to make the floor look a little less like a perfect mirror. I've chosen to create a uniform for this called `overrideColor` in the vertex shader:

```
uniform vec3 overrideColor;
...
Color = overrideColor * color;
```

And in the drawing code for the reflected cube

```
glUniform3f(uniColor, 0.3f, 0.3f, 0.3f);
glDrawArrays(GL_TRIANGLES, 0, 36);
glUniform3f(uniColor, 1.0f, 1.0f, 1.0f);
```

where `uniColor` is the return value of a `glGetUniformLocation` call.



Awesome! I hope that, especially in chapters like these, you get the idea that working with an API as low-level as OpenGL can be a lot of fun and pose interesting challenges! As usual, the final code is available [here](#).

Exercises

There are no real exercises for this chapter, but there are a lot more interesting effects that you can create with the stencil buffer. I'll leave researching the implementation of other effects, such as [stencil shadows](#) and [object outlining](#) as an exercise to you.

Framebuffers

In the previous chapters we've looked at the different types of buffers OpenGL offers: the color, depth and stencil buffers.

- These buffers occupy video memory like any other OpenGL object, but so far we've had little control over them besides specifying the pixel formats when you created the OpenGL context.
- This combination of buffers is known as the default framebuffer and as you've seen, a framebuffer is an area in memory that can be rendered to.
- What if you want to take a rendered result and do some additional operations on it, such as post-processing as seen in many modern games?

In this chapter we'll look at framebuffer objects, which are a means of creating additional framebuffers to render to.

- The great thing about framebuffers is that they allow you to render a scene directly to a texture, which can then be used in other rendering operations.
- After discussing how framebuffer objects work, I'll show you how to use them to do post-processing on the scene from the previous chapter.

Creating a new framebuffer

The first thing you need is a framebuffer object to manage your new framebuffer.

```
GLuint frameBuffer;  
glGenFramebuffers(1, &frameBuffer);
```

You cannot use this framebuffer yet at this point, because it is not *complete*. A framebuffer is generally complete if:

- At least one buffer has been attached (e.g. color, depth, stencil)
- There must be at least one color attachment (*OpenGL 4.1 and earlier*)
- All attachments are complete (*For example, a texture attachment needs to have memory reserved*)
- All attachments must have the same number of multi-samples

You can check if a framebuffer is complete at any time by calling `glCheckFramebufferStatus` and check if it returns `GL_FRAMEBUFFER_COMPLETE`.

- See the [reference](#) for other return values.
- You don't have to do this check, but it's usually a good thing to verify, just like checking if your shaders compiled successfully.

Now, let's bind the framebuffer to work with it.

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

The first parameter specifies the target the framebuffer should be attached to. OpenGL makes a distinction here between `GL_DRAW_FRAMEBUFFER` and `GL_READ_FRAMEBUFFER`.

- The framebuffer bound to read is used in calls to `glReadPixels`, but, since this distinction in normal applications is fairly rare, you can have your actions apply to both by using `GL_FRAMEBUFFER`.

```
glDeleteFramebuffers(1, &framebuffer);
```

Don't forget to clean up after you're done.

Attachments

Your framebuffer can only be used as a render target if memory has been allocated to store the results.

- This is done by attaching *images* for each buffer (color, depth, stencil or a combination of depth and stencil).
- There are two kinds of objects that can function as images: texture objects and **renderbuffer objects**.
- The advantage of the former is that they can be directly used in shaders as seen in the previous chapters, but **renderbuffer objects** may be more optimized specifically as render targets depending on your implementation.

Texture images

We'd like to be able to render a scene and then use the result in the color buffer in another rendering operation, so a texture is ideal in this case.

- Creating a texture for use as an image for the color buffer of the new framebuffer is as simple as creating any texture.

```
GLuint texColorBuffer;  
glGenTextures(1, &texColorBuffer);  
glBindTexture(GL_TEXTURE_2D, texColorBuffer);  
  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 800, 600, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);  
  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

The difference between this texture and the textures you've seen before is the `NULL` value for the data parameter.

- That makes sense, because the data is going to be created dynamically this time with rendering operations.
- Since this is the image for the color buffer, the `format` and `internalformat` parameters are a bit more restricted.
- The `format` parameter will typically be limited to either `GL_RGB` or `GL_RGBA` and the `internalformat` to the color formats.

I've chosen the default RGB internal format here, but you can experiment with more exotic formats like `GL_RGB10` if you want 10 bits of color precision.

- My application has a resolution of 800 by 600 pixels, so I've made this new color buffer match that.
- The resolution doesn't have to match the one of the default framebuffer, but don't forget a `glViewport` call if you do decide to vary.

The one thing that remains is attaching the image to the framebuffer.

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texColorBuffer, 0);
```

The second parameter implies that you can have multiple color attachments.

- A fragment shader can output different data to any of these by linking `out` variables to attachments with the `glBindFragDataLocation` function we used earlier. We'll stick to one output for now.
- The last parameter specifies the mipmap level the image should be attached to. Mipmapping is not of any use, since the color buffer image will be rendered at its original size when using it for post-processing.

Renderbuffer Object images

As we're using a depth and stencil buffer to render the spinning cube of cuteness, we'll have to create them as well.

OpenGL allows you to combine those into one image, so we'll have to create just one more before we can use the framebuffer. Although we could do this by creating another texture, it is more efficient to store these buffers in a Renderbuffer Object, because we're only interested in reading the color buffer in a shader.

```
GLuint rboDepthStencil;  
glGenRenderbuffers(1, &rboDepthStencil);  
glBindRenderbuffer(GL_RENDERBUFFER, rboDepthStencil);  
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, 800, 600);
```

Creating a **renderbuffer** object is very similar to creating a texture, the difference being is that this object is designed to be used as image instead of a general purpose data buffer like a texture.

I've chosen the **GL_DEPTH24_STENCIL8** internal format here, which is suited for holding both the depth and stencil buffer with 24 and 8 bits of precision respectively.

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rboDepthStencil);
```

Attaching it is easy as well. You can delete this object like any other object at a later time with a call to **glDeleteRenderbuffers**.

Using a framebuffer

Selecting a framebuffer as render target is very easy, in fact it can be done with a single call.

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

After this call, all rendering operations will store their result in the attachments of the newly created framebuffer. To switch back to the default framebuffer visible on your screen, **simply pass 0**.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Note that although only the default framebuffer will be visible on your screen, you can read any framebuffer that is currently bound with a call to `glReadPixels` as long as it's not only bound to `GL_DRAW_FRAMEBUFFER`.

Post-processing

In games nowadays post-processing effects seem almost just as important as the actual scenes being rendered on screen, and indeed some spectacular results can be accomplished with different techniques.

Post-processing effects in real-time graphics are commonly implemented in fragment shaders with the rendered scene as input in the form of a texture.

- Framebuffer objects allow us to use a texture to contain the color buffer, so we can use them to prepare input for a post-processing effect.

To use shaders to create a post-processing effect for a scene previously rendered to a texture, it is commonly rendered as a screen filling 2D rectangle. That way the original scene with the effect applied fills the screen at its original size as if it was rendered to the default framebuffer in the first place.

Of course, you can get creative with framebuffers and use them to do anything from portals to cameras in the game world by rendering a scene multiple times from different angles and display that on monitors or other objects in the final image. These uses are more specific, so I'll leave them as an exercise to you.

Changing the code

Unfortunately, it's a bit more difficult to cover the changes to the code step-by-step here, especially if you've strayed from the sample code here.

Now that you know how a framebuffer is created and bound however and with some care put into it, you should be able to do it.

Let's globally walk through the steps here.

- First try creating the framebuffer and checking if it is complete. Try binding it as render target and you'll see that your screen turns black because the scene is no longer rendered to the default framebuffer. Try changing the clear color of the scene and reading it back using `glReadPixels` to check if the scene renders properly to the new framebuffer.
- Next, try creating a new shader program, vertex array object and vertex buffer object to render things in 2D as opposed to 3D. It is useful to switch back to the default framebuffer for this to easily see your results. Your 2D shader shouldn't need transformation matrices. Try rendering a rectangle in front of the 3D spinning cube scene this way.
- Finally, try rendering the 3D scene to the framebuffer created by you and the rectangle to the default framebuffer. Now try using the texture of the framebuffer in the rectangle to render the scene.

I've chosen to have only 2 position coordinates and 2 texture coordinates for my 2D rendering. My 2D shaders look like this:

```
#version 150 core
in vec2 position;
in vec2 texcoord;
out vec2 Texcoord;
void main()
{
    Texcoord = texcoord;
    gl_Position = vec4(position, 0.0, 1.0);
}
```

```
#version 150 core
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFramebuffer;
void main()
{
    outColor = texture(texFramebuffer, Texcoord);
}
```

With this shader, the output of your program should be the same as before you even knew about framebuffers. Rendering a frame roughly looks like this:

```
// Bind our framebuffer and draw 3D scene (spinning cube)
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glBindVertexArray(vaoCube);
glEnable(GL_DEPTH_TEST);
glUseProgram(sceneShaderProgram);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texKitten);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texPuppy);

// Draw cube scene here

// Bind default framebuffer and draw contents of our framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindVertexArray(vaoQuad);
glDisable(GL_DEPTH_TEST);
glUseProgram(screenShaderProgram);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texColorBuffer);

glDrawArrays(GL_TRIANGLES, 0, 6);
```

The 3D and 2D drawing operations both have their own vertex array (cube versus quad), shader program (3D vs 2D post-processing) and textures. You can see that binding the color buffer texture is just as easy as binding regular textures.

Do mind that calls like `glBindTexture` which change the OpenGL state are relatively expensive, so try keeping them to a minimum.

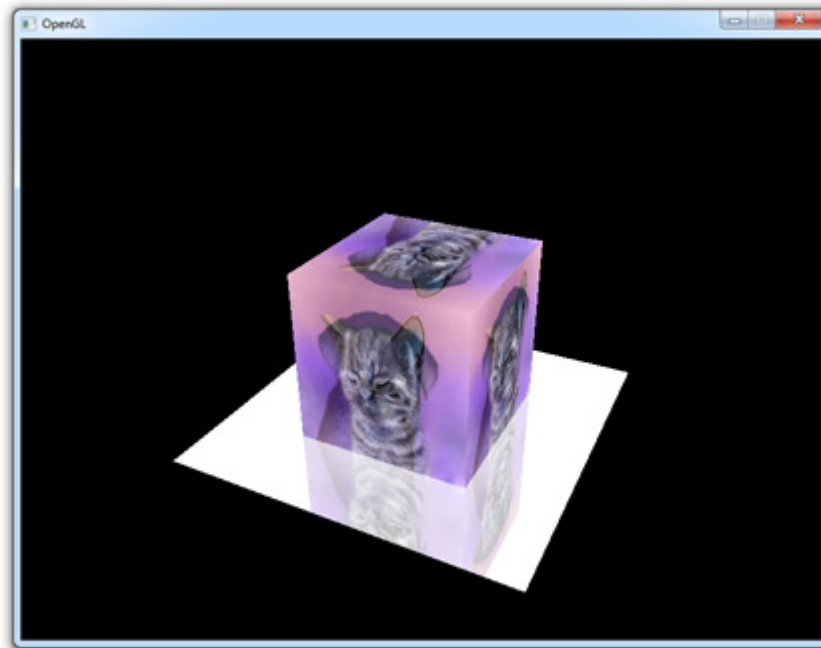
I think that no matter how well I explain the general structure of the program here, some of you just like to look at some new sample code and perhaps run a `diff` on it and the code from the previous chapter.

Post-processing effects

I will now discuss various interesting post-processing effects, how they work and what they look like.

Color manipulation

Inverting the colors is an option usually found in image manipulation programs, but you can also do it yourself using shaders!



As color values are floating point values ranging from `0.0` to `1.0`, inverting a channel is as simple as calculating `1.0 - channel`. If you do this for each channel (red, green, blue) you'll get an inverted color. In the fragment shader, that can be done like this.

```
outColor = vec4(1.0, 1.0, 1.0, 1.0) - texture(texFramebuffer, Texcoord);
```

This will also affect the alpha channel, but that doesn't matter because alpha blending is disabled by default.

Making colors grayscale can be naively done by calculating the average intensity of each channel.

```
outColor = texture(texFramebuffer, Texcoord);  
float avg = (outColor.r + outColor.g + outColor.b) / 3.0;  
outColor = vec4(avg, avg, avg, 1.0);
```

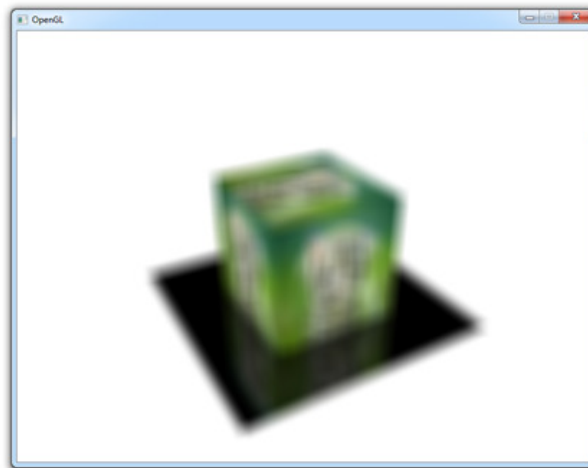
This works fine, but humans are the most sensitive to green and the least to blue, so a better conversion would work with weighed channels.

```
outColor = texture(texFramebuffer, Texcoord);  
float avg = 0.2126 * outColor.r + 0.7152 * outColor.g + 0.0722 * outColor.b;  
outColor = vec4(avg, avg, avg, 1.0);
```

Blur

There are two well known blur techniques: box blur and Gaussian blur.

The latter results in a higher quality result, but the former is easier to implement and still approximates Gaussian blur fairly well.



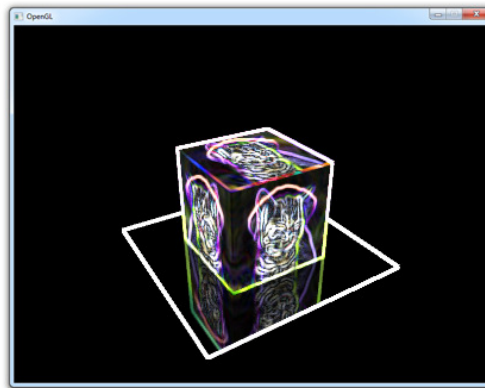
Blurring is done by sampling pixels around a pixel and calculating the average color.

```
const float blurSizeH = 1.0 / 300.0;
const float blurSizeV = 1.0 / 200.0;
void main()
{
    vec4 sum = vec4(0.0);
    for (int x = -4; x <= 4; x++)
        for (int y = -4; y <= 4; y++)
            sum += texture(
                texFramebuffer,
                vec2(Texcoord.x + x * blurSizeH, Texcoord.y + y * blurSizeV)
            ) / 81.0;
    outColor = sum;
}
```

You can see that a total amount of 81 samples is taken. You can change the amount of samples on the X and Y axes to control the amount of blur. The `blurSize` variables are used to determine the distance between each sample. A higher sample count and lower sample distance results in a better approximation, but also rapidly decreases performance, so try finding a good balance.

Sobel

The Sobel operator is often used in edge detection algorithms, let's find out what it looks like.



The fragment shader looks like this:

```
vec4 top      = texture(texFramebuffer, vec2(Texcoord.x, Texcoord.y + 1.0 / 200.0));
vec4 bottom   = texture(texFramebuffer, vec2(Texcoord.x, Texcoord.y - 1.0 / 200.0));
vec4 left     = texture(texFramebuffer, vec2(Texcoord.x - 1.0 / 300.0, Texcoord.y));
vec4 right    = texture(texFramebuffer, vec2(Texcoord.x + 1.0 / 300.0, Texcoord.y));
vec4 topLeft  = texture(texFramebuffer, vec2(Texcoord.x - 1.0 / 300.0, Texcoord.y + 1.0 / 200.0));
vec4 topRight = texture(texFramebuffer, vec2(Texcoord.x + 1.0 / 300.0, Texcoord.y + 1.0 / 200.0));
vec4 bottomLeft = texture(texFramebuffer, vec2(Texcoord.x - 1.0 / 300.0, Texcoord.y - 1.0 / 200.0));
vec4 bottomRight = texture(texFramebuffer, vec2(Texcoord.x + 1.0 / 300.0, Texcoord.y - 1.0 / 200.0));
vec4 sx = -topLeft - 2 * left - bottomLeft + topRight + 2 * right + bottomRight;
vec4 sy = -topLeft - 2 * top - topRight + bottomLeft + 2 * bottom + bottomRight;
vec4 sobel = sqrt(sx * sx + sy * sy);
outColor = sobel;
```

Just like the blur shader, a few samples are taken and combined in an interesting way. You can read more about the technical details elsewhere.

Conclusion

The cool thing about shaders is that you can manipulate images on a per-pixel basis in real time because of the immense parallel processing capabilities of your graphics card. It is no surprise that newer versions of software like Photoshop use the graphics card to accelerate image manipulation operations! There are many more complex effects like HDR, motion blur and SSAO (screen space ambient occlusion), but those involve a little more work than a single shader, so they're beyond the scope of this chapter.

Exercises

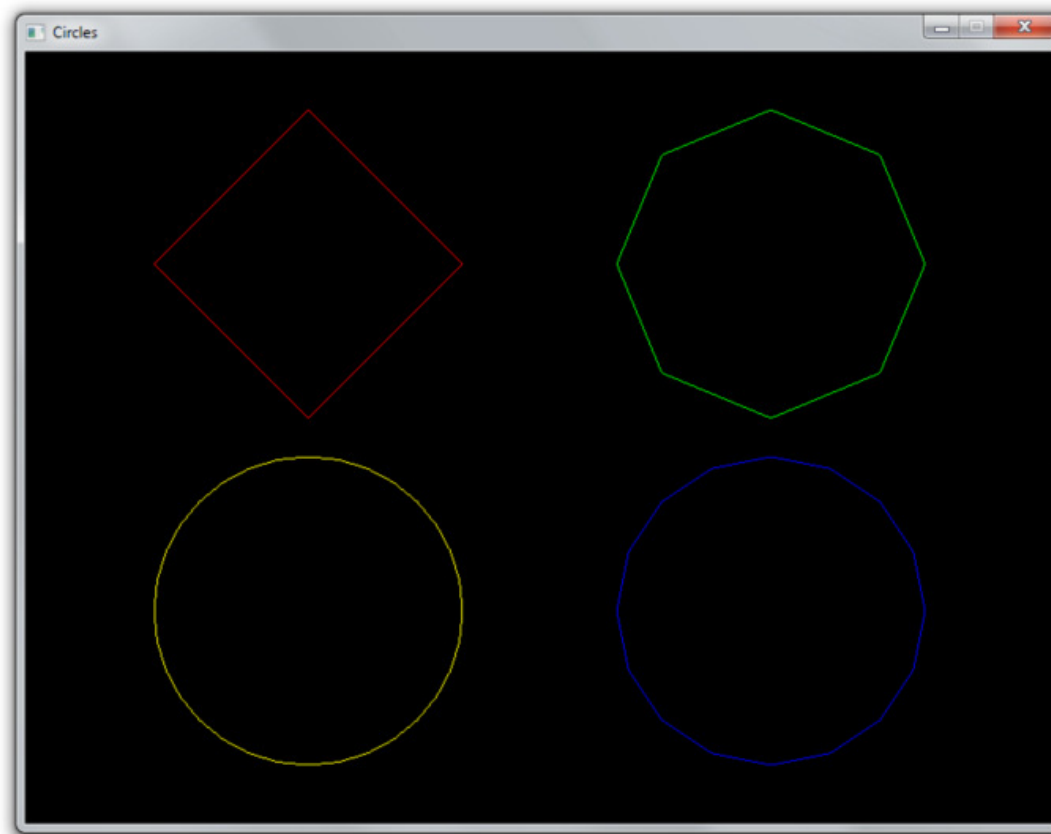
- Try implementing the two-pass Gaussian blur effect by adding another framebuffer. ([Solution](#))
- Try adding a panel in the 3D scene displaying that very scene from a different angle. ([Solution](#))

Geometry shaders

So far, we've used vertex and fragment shaders to manipulate our input vertices into pixels on the screen.

- Since OpenGL 3.2 there is a third optional type of shader that sits between the vertex and fragment shaders, known as the *geometry shader*.
- This shader has the unique ability to create new geometry on the fly using the output of the vertex shader as input.

Since we've neglected the kitten from the previous chapters for too long, it ran off to a new home. This gives us a good opportunity to start fresh. At the end of this chapter, we'll have the following demo:



That doesn't look all that exciting... until you consider that the result above was produced with a single draw call:

```
glDrawArrays(GL_POINTS, 0, 4);
```

Note that everything geometry shaders can do can be accomplished in other ways, but their ability to generate geometry from a small amount of input data allows you to reduce CPU -> GPU bandwidth usage.

Setup

Let's start by writing some simple code that just draws 4 red points to the screen.

```
// Vertex shader
const char* vertexShaderSrc = R"glsl(
    #version 150 core
    in vec2 pos;
    void main()
    {
        gl_Position = vec4(pos, 0.0, 1.0);
    }
)glsl";

// Fragment shader
const char* fragmentShaderSrc = R"glsl(
    #version 150 core
    out vec4 outColor;
    void main()
    {
        outColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
)glsl";
```

We'll start by declaring two very simple vertex and fragment shaders at the top of the file.

- The vertex shader simply forwards the position attribute of each point and the fragment shader always outputs red. Nothing special there.

Let's also add a helper function to create and compile a shader:

```
GLuint createShader(GLenum type, const GLchar* src) {  
    GLuint shader = glCreateShader(type);  
    glShaderSource(shader, 1, &src, nullptr);  
    glCompileShader(shader);  
    return shader;  
}
```

In the `main` function, create a window and OpenGL context with a library of choice and initialize GLEW. The shaders and compiled and activated:

```
GLuint vertexShader = createShader(GL_VERTEX_SHADER, vertexShaderSrc);  
GLuint fragmentShader = createShader(GL_FRAGMENT_SHADER, fragmentShaderSrc);  
GLuint shaderProgram = glCreateProgram();  
glAttachShader(shaderProgram, vertexShader);  
glAttachShader(shaderProgram, fragmentShader);  
glLinkProgram(shaderProgram);  
glUseProgram(shaderProgram);
```

After that, create a buffer that holds the coordinates of the points:

```
GLuint vbo;  
glGenBuffers(1, &vbo);  
float points[] = {  
    -0.45f, 0.45f,  
    0.45f, 0.45f,  
    0.45f, -0.45f,  
    -0.45f, -0.45f,  
};  
glBindBuffer(GL_ARRAY_BUFFER, vbo);  
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
```

We have 4 points here, each with x and y device coordinates. Remember that device coordinates range from -1 to 1 from left to right and bottom to top of the screen, so each corner will have a point.

Then create a VAO and set the vertex format specification:

```
// Create VAO
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// Specify layout of point data
GLint posAttrib = glGetAttribLocation(shaderProgram, "pos");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

And finally the render loop:

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

glDrawArrays(GL_POINTS, 0, 4);
```

With this code, you should now see 4 red points on a black background as shown below: If you are having problems, have a look at the [reference source code](#).

Basic geometry shader

To understand how a geometry shader works, let's look at an example:

```
#version 150 core
layout(points) in;
layout(line_strip, max_vertices = 2) out;
void main()
{
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Input types

Whereas a vertex shader processes vertices and a fragment shader processes fragments, a geometry shader processes entire primitives. The first line describes what kind of primitives our shader should process.

```
layout(points) in;
```

The available types are listed below, along with their equivalent drawing command types:

- points - GL_POINTS (1 vertex)
- lines - GL_LINES, GL_LINE_STRIP, GL_LINE_LIST (2 vertices)
- lines_adjacency - GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY (4 vertices)
- triangles - GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN (3 vertices)
- triangles_adjacency - GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY (6 vertices)

Since we're drawing `GL_POINTS`, the `points` type is appropriate.

Output types

The next line describes the output of the shader.

What's interesting about geometry shaders is that they can output an entirely different type of geometry and the number of generated primitives can even vary!

```
layout(line_strip, max_vertices = 2) out;
```

The second line specifies the output type and the maximum amount of vertices it can pass on.

This is the maximum amount for the shader invocation, not for a single primitive (`line_strip` in this case).

The following output types are available:

- points
- line_strip
- triangle_strip

These types seem somewhat restricted, but if you think about it, these types are sufficient to cover all possible types of primitives.

For example, a triangle_strip with only 3 vertices is equivalent to a regular triangle.

Vertex input

The `gl_Position`, as set in the vertex shader, can be accessed using the `gl_in` array in the geometry shader. It is an array of structs that looks like this:

```
in gl_PerVertex
{
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

Notice that vertex attributes like `pos` and `color` are not included, we'll look into accessing those later.

Vertex output

The geometry shader program can call two special functions to generate primitives, `EmitVertex` and `EndPrimitive`. Each time the program calls `EmitVertex`, a vertex is added to the current primitive. When all vertices have been added, the program calls `EndPrimitive` to generate the primitive.

```
void main()
{
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Before calling `EmitVertex`, the attributes of the vertex should be assigned to variables like `gl_Position`, just like in the vertex shader. We'll look at setting attributes like `color` for the fragment shader later.

Now that you know the meaning of every line, can you explain what this geometric shader does?

It creates a single horizontal line for each point coordinate passed to it.

Creating a geometry shader

There's not much to explain, geometry shaders are created and activated in exactly the same way as other types of shaders. Let's add a geometry shader to our 4-point sample that doesn't do anything yet.

```
const char* geometryShaderSrc = R"glsl(  
    #version 150 core  
  
    layout(points) in;  
    layout(points, max_vertices = 1) out;  
  
    void main()  
    {  
        gl_Position = gl_in[0].gl_Position;  
        EmitVertex();  
        EndPrimitive();  
    }  
)glsl";
```

This geometry shader should be fairly straightforward. For each input point, it generates one equivalent output point. This is the minimum amount of code necessary to still display the points on the screen.

With the helper function, creating a geometry shader is easy:

```
GLuint geometryShader = createShader(GL_GEOMETRY_SHADER, geometryShaderSrc);
```

There's nothing special about attaching it to the shader program either:

```
glAttachShader(shaderProgram, geometryShader);
```

When you run the program now, it should still display the points as before. You can verify that the geometry shader is now doing its work by removing the code from its `main` function. You'll see that no points are being drawn anymore, because none are being generated!

Now, try replacing the geometry shader code with the line strip generating code from the previous section:

```
#version 150 core

layout(points) in;
layout(line_strip, max_vertices = 2) out;

void main()
{
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Even though we've made no changes to our draw call, the GPU is suddenly drawing tiny lines instead of points! Try experimenting a bit to get a feel for it. For example, try outputting rectangles by using `triangle_strip`.

Geometry shaders and vertex attributes

Let's add some variation to the lines that are being drawn by allowing each of them to have a unique color. By adding a color input variable to the vertex shader, we can specify a color per vertex and thus per generated line.

```
#version 150 core
in vec2 pos;
in vec3 color;
out vec3 vColor; // Output to geometry (or fragment) shader

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

Update the vertex specification in the program code:

```
GLuint posAttrib = glGetAttribLocation(shaderProgram, "pos");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), 0);

GLuint colAttrib = glGetAttribLocation(shaderProgram, "color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*) (2 * sizeof(float)));
```

And update the point data to include an RGB color per point:

```
float points[] = {
    -0.45f, 0.45f, 1.0f, 0.0f, 0.0f, // Red point
    0.45f, 0.45f, 0.0f, 1.0f, 0.0f, // Green point
    0.45f, -0.45f, 0.0f, 0.0f, 1.0f, // Blue point
    -0.45f, -0.45f, 1.0f, 1.0f, 0.0f, // Yellow point
};
```

Because the vertex shader is now not followed by a fragment shader, but a geometry shader, we have to handle the `vColor` variable as input there.

```
#version 150 core

layout(points) in;
layout(line_strip, max_vertices = 2) out;

in vec3 vColor[]; // Output from vertex shader for each vertex

out vec3 fColor; // Output to fragment shader

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

You can see that it is very similar to how inputs are handled in the fragment shader. The only difference is that inputs must be arrays now, because the geometry shader can receive primitives with multiple vertices as input, each with its own attribute values.

Because the color needs to be passed further down to the fragment shader, we add it as output of the geometry shader. We can now assign values to it, just like we did earlier with `gl_Position`.

```
void main()
{
    fColor = vColor[0]; // Point has only one vertex

    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.1, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.1, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```


Whenever `EmitVertex` is called now, a vertex is emitted with the current value of `fColor` as color attribute. We can now access that attribute in the fragment shader:

```
#version 150 core
in vec3 fColor;
out vec4 outColor;
void main()
{
    outColor = vec4(fColor, 1.0);
}
```

So, when you specify an attribute for a vertex, it is first passed to the vertex shader as input.

- The vertex shader can then choose to output it to the geometry shader.
- And then the geometry shader can choose to further output it to the fragment shader.

However, this demo is not very interesting. We could easily replicate this behavior by creating a vertex buffer with a single line and issuing a couple of draw calls with different colors and positions set with uniform variables.

Dynamically generating geometry

The real power of geometry shader lies in the ability to generate a varying amount of primitives, so let's create a demo that properly abuses this ability.

- Let's say you're making a game where the world consists of circles.
- You could draw a single model of a circle and repeatedly draw it, but this approach is not ideal.
- If you're too close, these "circles" will look like ugly polygons and if you're too far away, your graphics card is wasting performance on rendering complexity you can't even see.

We can do better with geometry shaders!

- We can write a shader that generates the appropriate resolution circle based on run-time conditions.
- Let's first modify the geometry shader to draw a 10-sided polygon at each point.
- If you remember your trigonometry, it should be a piece of cake:

```

#version 150 core
layout(points) in;
layout(line_strip, max_vertices = 11) out;

in vec3 vColor[];
out vec3 fColor;
const float PI = 3.1415926;

void main()
{
    fColor = vColor[0];
    for (int i = 0; i <= 10; i++) {
        // Angle between each side in radians
        float ang = PI * 2.0 / 10.0 * i;

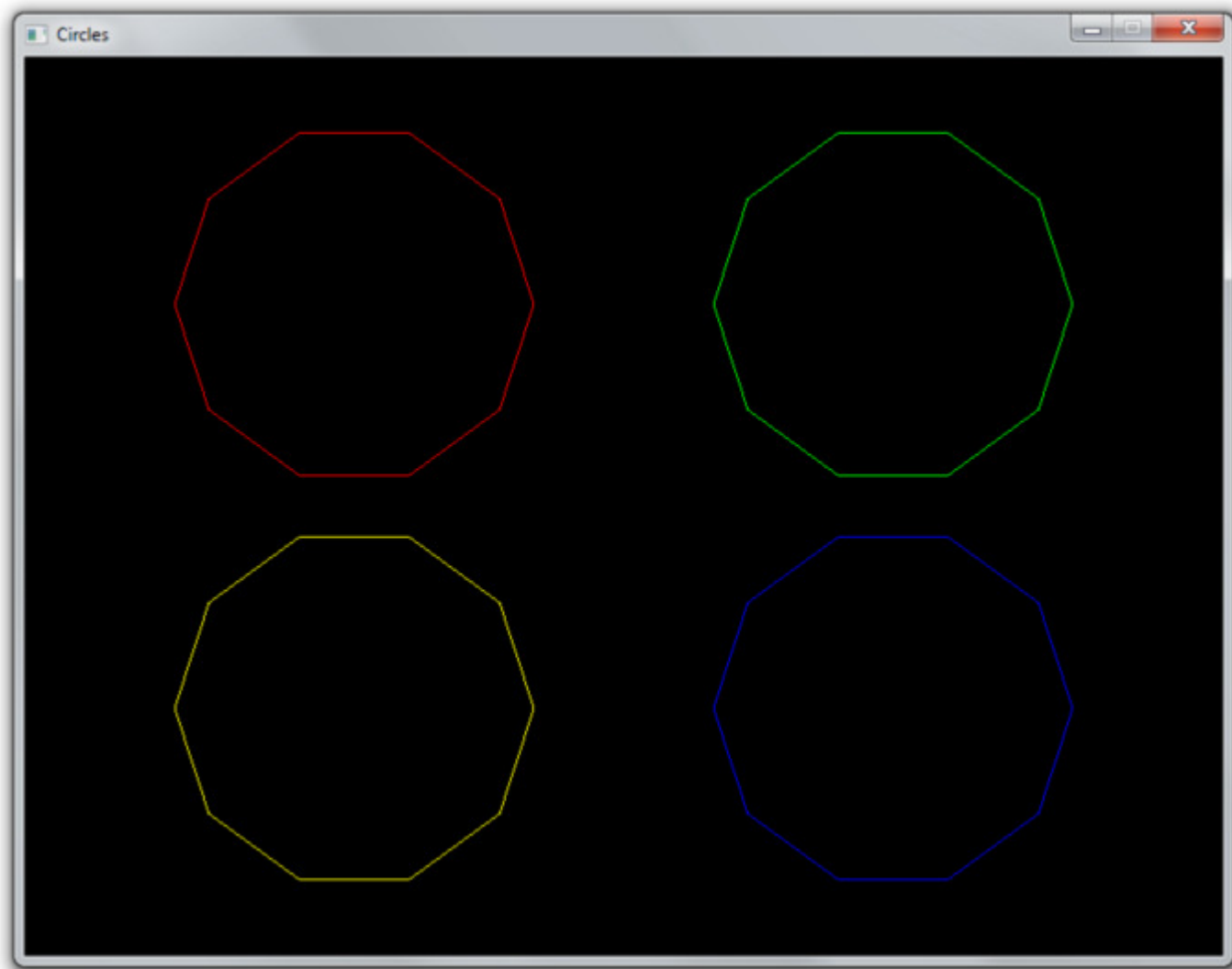
        // Offset from center of point (0.3 to accomodate for aspect ratio)
        vec4 offset = vec4(cos(ang) * 0.3, -sin(ang) * 0.4, 0.0, 0.0);
        gl_Position = gl_in[0].gl_Position + offset;

        EmitVertex();
    }

    EndPrimitive();
}

```

The first point is repeated to close the line loop, which is why 11 vertices are drawn. The result is as expected:



It is now trivial to add a vertex attribute to control the amount of sides. Add the new attribute to the data and to the specification:

```
float points[] = {  
    // Coordinates Color Sides  
    -0.45f, 0.45f, 1.0f, 0.0f, 0.0f, 4.0f,  
    0.45f, 0.45f, 0.0f, 1.0f, 0.0f, 8.0f,  
    0.45f, -0.45f, 0.0f, 0.0f, 1.0f, 16.0f,  
    -0.45f, -0.45f, 1.0f, 1.0f, 0.0f, 32.0f  
};  
...  
// Specify layout of point data  
GLint posAttrib = glGetAttribLocation(shaderProgram, "pos");  
glEnableVertexAttribArray(posAttrib);  
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);  
  
GLint colAttrib = glGetAttribLocation(shaderProgram, "color");  
glEnableVertexAttribArray(colAttrib);  
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*) (2 * sizeof(float)));  
  
GLint sidesAttrib = glGetAttribLocation(shaderProgram, "sides");  
glEnableVertexAttribArray(sidesAttrib);  
glVertexAttribPointer(sidesAttrib, 1, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*) (5 * sizeof(float)));
```

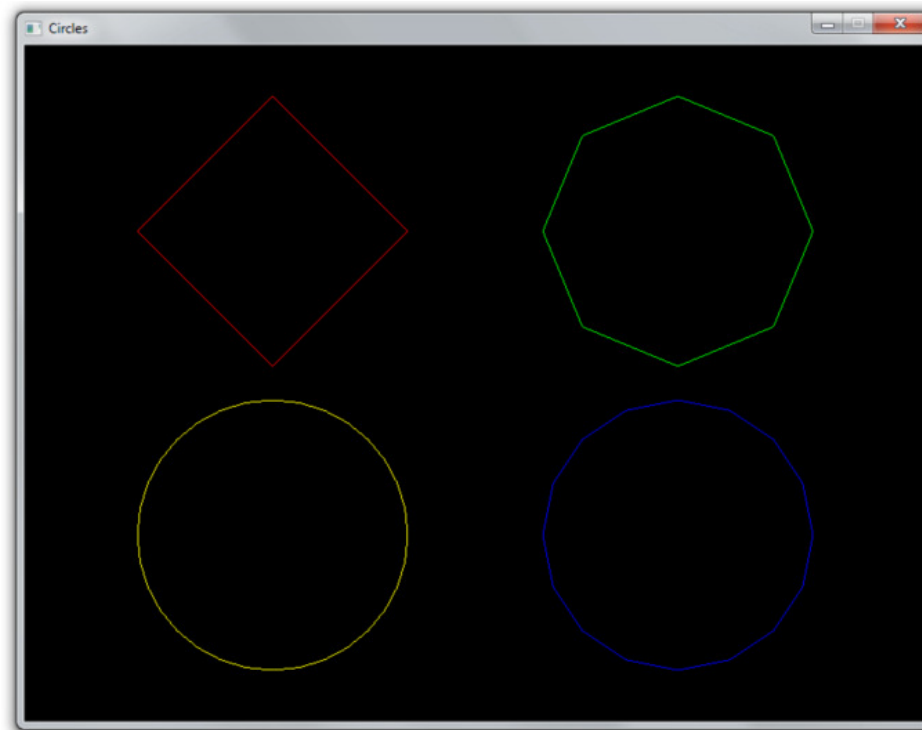
Alter the vertex shader to pass the value to the geometry shader:

```
#version 150 core  
in vec2 pos;  
in vec3 color;  
in float sides;  
out vec3 vColor;  
out float vSides;  
  
void main()  
{  
    gl_Position = vec4(pos, 0.0, 1.0);  
    vColor = color;  
    vSides = sides;  
}
```

And use the variable in the geometry shader instead of the magic number of sides `10.0`. It's also necessary to set an appropriate `max_vertices` value for our input, otherwise the circles with more vertices will be cut off.

```
layout(line_strip, max_vertices = 64) out;  
...  
in float vSides[];  
...  
// Safe, floats can represent small integers exactly  
for (int i = 0; i <= vSides[0]; i++) {  
    // Angle between each side in radians  
    float ang = PI * 2.0 / vSides[0] * i;  
    ...  
}
```

You can now create a circles with any amount of sides you desire by simply adding more points!



Without a geometry shader, we'd have to rebuild the entire vertex buffer whenever any of these circles have to change, now we can simply change the value of a vertex attribute. In a game setting, this attribute could be changed based on player distance as described above. You can find the full code [here](#).

Conclusion

Granted, geometry shaders may not have as many real world use cases as things like framebuffers and textures have, but they can definitely help with creating content on the GPU as shown here.

If you need to repeat a single mesh many times, like a cube in a voxel game, you could create a geometry shader that generates cubes from points in a similar fashion. However, for these cases where each generated mesh is exactly the same, there are more efficient methods like [instancing](#).

Lastly, with regards to portability, the latest WebGL and OpenGL ES standards do not yet support geometry shaders, so keep that in mind if you're considering the development of a mobile or web application.

Exercises

- Try using a geometry shader in a 3D scenario to create more complex meshes like cubes from points. ([Solution](#))

Transform feedback

Up until now we've always sent vertex data to the graphics processor and only produced drawn pixels in framebuffers in return. What if we want to retrieve the vertices after they've passed through the vertex or geometry shaders? In this chapter we'll look at a way to do this, known as **transform feedback**.

So far, we've used **VBOs (Vertex Buffer Objects)** to store vertices to be used for drawing operations.

- The transform feedback extension allows shaders to write vertices back to these as well.
- You could for example build a vertex shader that simulates gravity and writes updated vertex positions back to the buffer.

This way you don't have to transfer this data back and forth from graphics memory to main memory. On top of that, you get to benefit from the vast parallel processing power of today's GPUs.

Basic feedback

We'll start from scratch so that the final program will clearly demonstrate how simple transform feedback is. Unfortunately, there's no preview this time, because we're not going to draw anything in this chapter! Although this feature can be used to simplify effects like particle simulation, explaining these is a bit beyond the scope of these articles.

After you've understood the basics of transform feedback, you'll be able to find and understand plenty of articles around the web on these topics.

Let's start with a simple vertex shader.

```
const GLchar* vertexShaderSrc = R"glsl(  
    in float inValue;  
    out float outValue;  
  
    void main()  
    {  
        outValue = sqrt(inValue);  
    }  
)glsl";
```

This vertex shader does not appear to make much sense. It doesn't set a `gl_Position` and it only takes a single arbitrary float as input. Luckily, we can use transform feedback to capture the result, as we'll see momentarily.

```
GLuint shader = glCreateShader(GL_VERTEX_SHADER);  
glShaderSource(shader, 1, &vertexShaderSrc, nullptr);  
glCompileShader(shader);
```

```
GLuint program = glCreateProgram();  
glAttachShader(program, shader);
```

Compile the shader, create a program and attach the shader, but don't call `glLinkProgram` yet!

- Before linking the program, we have to tell OpenGL which output attributes we want to capture into a buffer.

```
const GLchar* feedbackVaryings[] = { "outValue" };  
glTransformFeedbackVaryings(program, 1, feedbackVaryings, GL_INTERLEAVED_ATTRIBS);
```

The first parameter is self-explanatory, the second and third parameter specify the length of the output names array and the array itself, and the final parameter specifies how the data should be written.

The following two formats are available:

- `GL_INTERLEAVED_ATTRIBS`: Write all attributes to a single buffer object.
- `GL_SEPARATE_ATTRIBS`: Writes attributes to multiple buffer objects or at different offsets into a buffer.

Sometimes it is useful to have separate buffers for each attribute, but let's keep it simple for this demo.

Now that you've specified the output variables, you can link and activate the program.

That is because the linking process depends on knowledge about the outputs.

```
glLinkProgram(program);  
glUseProgram(program);
```


After that, create and bind the VAO:

```
GLuint vao;  
glGenVertexArrays(1, &vao);  
glBindVertexArray(vao);
```

Now, create a buffer with some input data for the vertex shader:

```
GLfloat data[] = { 1.0f, 2.0f, 3.0f, 4.0f, 5.0f };  
  
GLuint vbo;  
glGenBuffers(1, &vbo);  
glBindBuffer(GL_ARRAY_BUFFER, vbo);  
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
```

The numbers in `data` are the numbers we want the shader to calculate the square root of and transform feedback will help us get the results back.

With regards to vertex pointers, you know the drill by now:

```
GLint inputAttrib = glGetAttribLocation(program, "inValue");  
glEnableVertexAttribArray(inputAttrib);  
glVertexAttribPointer(inputAttrib, 1, GL_FLOAT, GL_FALSE, 0, 0);
```

Transform feedback will return the values of `outValue`, but first we'll need to create a VBO to hold these, just like the input vertices:

```
GLuint tbo;  
glGenBuffers(1, &tbo);  
glBindBuffer(GL_ARRAY_BUFFER, tbo);  
glBufferData(GL_ARRAY_BUFFER, sizeof(data), nullptr, GL_STATIC_READ);
```

Notice that we now pass a `nullptr` to create a buffer big enough to hold all of the resulting floats, but without specifying any initial data. The appropriate usage type is now `GL_STATIC_READ`, which indicates that we intend OpenGL to write to this buffer and our application to read from it. (See [reference](#) for usage types)

We've now made all preparations for the **rendering** computation process. As we don't intend to draw anything, the rasterizer should be disabled:

```
glEnable(GL_RASTERIZER_DISCARD);
```

To actually bind the buffer we've created above as transform feedback buffer, we have to use a new function called `glBindBufferBase`.

```
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, tbo);
```

The first parameter is currently required to be `GL_TRANSFORM_FEEDBACK_BUFFER` to allow for future extensions. The second parameter is the index of the output variable, which is simply `0` because we only have one. The final parameter specifies the buffer object to bind.

Before doing the draw call, you have to enter transform feedback mode:

```
glBeginTransformFeedback(GL_POINTS);
```

It certainly brings back memories of the old `glBegin` days! Just like the geometry shader in the last chapter, the possible values for the primitive mode are a bit more limited.

- `GL_POINTS` — `GL_POINTS`
- `GL_LINES` — `GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`, `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`
- `GL_TRIANGLES` — `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES_ADJACENCY`, `GL_TRIANGLE_STRIP_ADJACENCY`

If you only have a vertex shader, as we do now, the primitive *must* match the one being drawn:

```
glDrawArrays(GL_POINTS, 0, 5);
```

Even though we're now working with data, the single numbers can still be seen as separate "points", so we use that primitive mode.

End the transform feedback mode:

```
glEndTransformFeedback();
```

Normally, at the end of a drawing operation, we'd swap the buffers to present the result on the screen. We still want to make sure the rendering operation has finished before trying to access the results, so we flush OpenGL's command buffer:

```
glFlush();
```

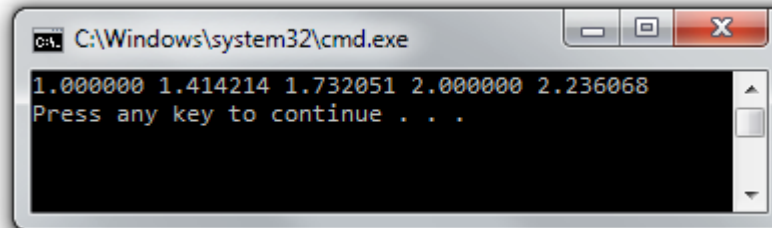
Getting the results back is now as easy as copying the buffer data back to an array:

```
GLfloat feedback[5];
```

```
glGetBufferSubData(GL_TRANSFORM_FEEDBACK_BUFFER, 0, sizeof(feedback), feedback);
```

If you now print the values in the array, you should see the square roots of the input in your terminal:

```
printf("%f %f %f %f %f\n", feedback[0], feedback[1], feedback[2], feedback[3], feedback[4]);
```



Congratulations, you now know how to make your GPU perform general purpose tasks with vertex shaders! Of course a real GPGPU framework like OpenCL is generally better at this, but the advantage of transform feedback is that you can directly repurpose the data in drawing operations, by for example binding the transform feedback buffer as array buffer and performing normal drawing calls.

If you have a graphics card and driver that supports it, you could also use compute shaders in OpenGL 4.3 instead, which were actually designed for tasks that are less related to drawing.

You can find the full code [here](#).

Feedback transform and geometry shaders

When you include a geometry shader, the transform feedback operation will capture the outputs of the geometry shader instead of the vertex shader. For example:

```
// Vertex shader
const GLchar* vertexShaderSrc = R"glsl(

    in float inValue;
    out float geoValue;

    void main()
    {
        geoValue = sqrt(inValue);
    }
)glsl";

// Geometry shader
const GLchar* geoShaderSrc = R"glsl(
    layout(points) in;
    layout(triangle_strip, max_vertices = 3) out;

    in float[] geoValue;
    out float outValue;

    void main()
    {
        for (int i = 0; i < 3; i++) {
            outValue = geoValue[0] + i;
            EmitVertex();
        }

        EndPrimitive();
    }
)glsl";
```

The geometry shader takes a point processed by the vertex shader and generates 2 more to form a triangle with each point having a 1 higher value.

```
GLuint geoShader = glCreateShader(GL_GEOMETRY_SHADER);
glShaderSource(geoShader, 1, &geoShaderSrc, nullptr);
glCompileShader(geoShader);

...

glAttachShader(program, geoShader);
```

Compile and attach the geometry shader to the program to start using it.

```
const GLchar* feedbackVaryings[] = { "outValue" };
glTransformFeedbackVaryings(program, 1, feedbackVaryings, GL_INTERLEAVED_ATTRIBS);
```

Although the output is now coming from the geometry shader, we've not changed the name, so this code remains unchanged. Because each input vertex will generate 3 vertices as output, the transform feedback buffer now needs to be 3 times as big as the input buffer:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(data) * 3, nullptr, GL_STATIC_READ);
```

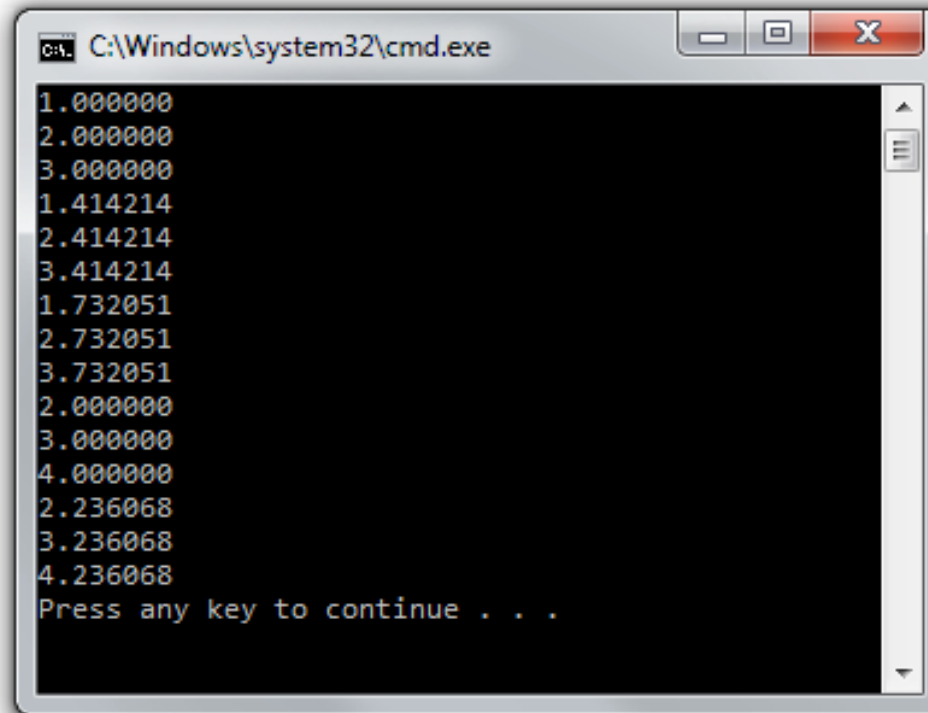
When using a geometry shader, the primitive specified to `glBeginTransformFeedback` must match the output type of the geometry shader:

```
glBeginTransformFeedback(GL_TRIANGLES);
```

Retrieving the output still works the same:

```
// Fetch and print results
GLfloat feedback[15];
glGetBufferSubData(GL_TRANSFORM_FEEDBACK_BUFFER, 0, sizeof(feedback), feedback);

for (int i = 0; i < 15; i++) {
    printf("%f\n", feedback[i]);
}
```



```
C:\Windows\system32\cmd.exe
1.000000
2.000000
3.000000
1.414214
2.414214
3.414214
1.732051
2.732051
3.732051
2.000000
3.000000
4.000000
2.236068
3.236068
4.236068
Press any key to continue . . .
```

Although you have to pay attention to the feedback primitive type and the size of your buffers, adding a geometry shader to the equation doesn't change much other than the shader responsible for output.

The full code can be found [here](#).

Variable feedback

As we've seen in the previous chapter, geometry shaders have the unique property to generate a variable amount of data. Luckily, there are ways to keep track of how many primitives were written by using *query objects*.

Just like all the other objects in OpenGL, you'll have to create one first:

```
GLuint query;
glGenQueries(1, &query);
```

Then, right before calling `glBeginTransformFeedback`, you have to tell OpenGL to keep track of the number of primitives written:

```
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, query);
```

After `glEndTransformFeedback`, you can stop "recording":

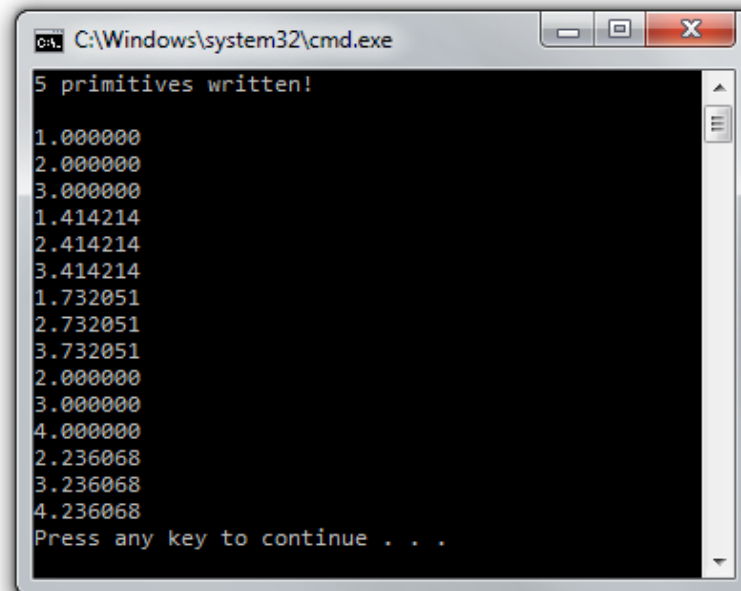
```
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);
```

Retrieving the result is done as follows:

```
GLuint primitives;  
glGetQueryObjectuiv(query, GL_QUERY_RESULT, &primitives);
```

You can then print that value along with the other data:

```
printf("%u primitives written!\n\n", primitives);
```



```
C:\Windows\system32\cmd.exe  
5 primitives written!  
1.000000  
2.000000  
3.000000  
1.414214  
2.414214  
3.414214  
1.732051  
2.732051  
3.732051  
2.000000  
3.000000  
4.000000  
2.236068  
3.236068  
4.236068  
Press any key to continue . . .
```

Notice that it returns the number of primitives, not the number of vertices. Since we have 15 vertices, with each triangle having 3, we have 5 primitives.

Query objects can also be used to record things such as `GL_PRIMITIVES_GENERATED` when dealing with just geometry shaders and `GL_TIME_ELAPSED` to measure time spent on the server (graphics card) doing work.

See [the full code](#) if you got stuck somewhere on the way.

Conclusion

You now know enough about geometry shaders and transform feedback to make your graphics card do some very interesting work besides just drawing! You can even combine transform feedback and rasterization to update vertices and draw them at the same time!

Exercises

- Try writing a vertex shader that simulates gravity to make points hover around the mouse cursor using transform feedback to update the vertices. ([Solution](#))

Vid#1 – Welcome to OpenGL:

- **What is OpenGL?**
 - OpenGL is a Graphics API to call over a thousand graphics functions for GUI applications by accessing the GPU (Graphics Processing Unit)
 - OpenGL, DirectX12, Direct3D11, Vulkan, Metal are all different Graphics APIs designed to provide graphics functions that control various GPU hardware of many Graphics manufacturers (Mfgs.).
 - OpenGL IS NOT a library, engine, framework or implementation (NO Code)!
 - OpenGL at its core IS JUST a C++ graphics specification that defines functions and their input parameters, expected outputs, and return values.
 - Each GPU manufacturer implements its own Code of functions that support the OpenGL specification.
 - Example: Nvidia wrote its own code OpenGL Drivers to support its version(s) of OpenGL
 - Every GPU Mfg. will supply their own version(s) of OpenGL Drivers to support their hardware implementations
 - Therefore, OpenGL IS NOT technically open source!!! Mfgs., Generally DO NOT share the same source code, each develops their own implementation(s).
 - However, there are some OpenGL source code implementation(s) available for open source hardware developers made available through GitHub.
 - The OpenGL specification was created to be cross-platform, is the easiest Graphics API to learn today, and is the most stable today.
- **Legacy OpenGL (1990's) vs. Modern OpenGL (**
 - Legacy OpenGL – limited functionality, control and code that works on a set of presets which are enabled = true or disabled = false.
 - Modern OpenGL – increased functionality, control and code that implements programmable Shaders (code that runs on the faster GPU than the slower CPU)
 - Example: GPU running complex lighting code algorithms
- **How we can use OpenGL?**
 - Write OpenGL implementation in C++ (arguably the best language for the task)
 - Use Visual Studio IDE Tool Set on Windows to learn OpenGL
- **What will this series will cover?**
 - Fast 2D Graphics, Batching, and Fast 2D Rendering
 - Fast 3D Graphics -
 - Implement Lighting, Shadows, Deferred Rendering, Physically based Rendering
 - Screens based plume, reflection, etc. with many examples

Vid#2 – Setting up OpenGL and Creating a Window in C++

- Create an operating system specific Window using the target platform's Window API
 - For Windows 10, create a Window using the Win32 API OR
 - Use the OpenGL GLFW lightweight Library that provides the appropriate platform layer implementation code to create windows in Windows, Mac, and Linux.
 - GLFW Library will allow us to create a window, create an OpenGL context, and provide access to some basic elements like Input without a framework making it platform independent.
 - Download GLFW 3.3.2 (Released on January 20, 2020) at www.glfw.org
 - Get prebuilt binaries for (32-bit or 64-bit) target Windows machine: CLK → Download > 32-bit Windows Binaries > Open > Extract All > New Folder: C:\Documents\The Chernobyl OpenGL > Select Folder > Extract > created new folder: glfw-3.3.2.bin.WIN32.
 - Review and Copy the sample GLFW code under Documentation
- VS2019 Create New Project:
 - File > New > Empty Project (C++ Windows Console) > Next > Create
 - Project Name: OpenGL
 - Location: C:\Dev\Cherno
 - Solution Name: OpenGL
 - Change Project Properties Settings:
 - RT+CLK > NewWinProject > Project > Properties (Alt + Enter)
 - Configurations: Active(Debug) ▢ All Configurations
 - Platform: Active(Win32) ▢ All Platforms
 - Configuration Properties > General
 - Output Directory: <different options> → \$(SolutionDir)bin\\$(Platform)\\$(Configuration)\ > Edit > Apply > Ok
 - Intermediate Directory: <different options> → \$(SolutionDir)bin\intermediates\\$(Platform)\\$(Configuration)\ > Edit > Apply > Ok
 - Target Name: \$(ProjectName)
 - Linker > General
 - Output File: \$(OutDir)\\$(TargetName)\\$(TargetExt)
 - Add new folder: src and Add new item (main.cpp) in this folder
 - CLK > OpenGL > CLK icon: Show All Files (mimics Windows Explorer files structure)
 - RT+CLK > OpenGL > Add > New Folder: src
 - RT+CLK > src > Add > New Item: C++ File (.cpp)
 - Name: Application.cpp
 - Location: C:\Dev\Cherno\OpenGL\OpenGL\src → Add

– Initial Application.cpp Test Code → Passed

```
#include <iostream>
int main()
{
    std::cout << "Hello" << std::endl;
    std::cin.get();
}
```

▪ GLFW Documentation – Example Code → copy, paste and replace Initial Test Code in Application.cpp

```
#include <GLFW/glfw3.h>
int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */
    if (!glfwInit())
        return -1;

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }
    /* Make the window's context current */
    glfwMakeContextCurrent(window);

    /* Loop until the user closes the window */
    while (!glfwWindowShouldClose(window))
    {
        /* Render here */
        glClear(GL_COLOR_BUFFER_BIT);

        /* Swap front and back buffers */
        glfwSwapBuffers(window);
    }
}
```

```

    /* Poll for and process events */
    glfwPollEvents();
}

glfwTerminate();
return 0;
}

```

▪ How to setup GLFW in our OpenGL project:

- Downloaded zip extracted to: C:\Documents\The Chernobyl\glfw-3.3.2.bin.WIN32
- Two(2) required folders: include, lib-vc2019 for this project
 - Lib-vc2019 folder:
 - glfw3.dll – used for linking at runtime (can be deleted)
 - glfw3.lib – used in this project for static linking
 - In the OpenGL Solutions directory, create a new folder: Dependencies
 - In the Dependencies folder, create new folder: GLFW
 - In the GLFW folder, copy & paste the include folder and lib-vc2019 folder and their files
 - glfw3dll.lib – used for dynamic linking (can be deleted)
 - include folder:
 - glfw3.h
 - glfw3native.h
- Configure Properties (Alt + Enter)
 - Configurations: All Configurations
 - Platforms: All Platforms
 - C++ > General
 - Add GLFW include folder to project
 - Additional Include Directories: \$(SolutionDir)Dependencies\GLFW\include → **Apply**
 - Check #include <GLFW/glfw3.h> no longer has any errors in Application.cpp
 - Linker > General
 - Add glfw3.lib to linker
 - Additional Library Directories: \$(SolutionDir)Dependencies\GLFW\lib-vc2015 → **Apply**
 - Linker > Input
 - Additional Dependencies
 - Delete ALL existing Additional Dependencies
 - Add “glfw3.lib” to new Additional Dependencies → **Apply** → **Ok**

- Check ALL Application.cpp code has no warnings and Build project (Ctrl + Shift + B) – Build Failed with many Linker Errors

– Fix Build Errors (14:00)

- Error LNK2019 unresolved external symbol __imp__glClear@4 referenced in function _main
 - Solution: link to OpenGL library function
 - Add semi-colon followed by “opengl32.lib” to new Additional Dependencies → **Apply** → **Ok**
 - Rebuild and check that this error has gone away – Build Failed with more Linker Errors
- Error LNK2019 error LNK2019: unresolved external symbol __imp__RegisterDeviceNotificationW@12 referenced in function _createHelperWindow
 - Solution: link to OpenGL library function
 - Google: RegisterDeviceNotificationW in MSDN Docs, scroll down to Requirements and copy Library: User32.lib
 - Add semi-colon followed by “User32.lib” to new Additional Dependencies → **Apply** → **Ok**
 - Rebuild and check that this error has gone away – Build Failed with more Linker Errors
- Error LNK2019 error LNK2019: unresolved external symbol __imp__CreateDCW@16 referenced in function __glfwPlatformGetGammaRamp
 - Solution: link to OpenGL library function
 - Google: CreateDCW in MSDN Docs, scroll down to Requirements and copy Library: Gdi32.lib
 - Add semi-colon followed by “Gdi32.lib” to new Additional Dependencies → **Apply** → **Ok**
 - Rebuild and check that this error has gone away – Build Failed with more Linker Errors
- Error LNK2019: unresolved external symbol __imp__DragQueryFileW@16 referenced in function _windowProc@16
 - Solution: link to OpenGL library function
 - Google: DragQueryFileW in MSDN Docs, scroll down to Requirements and copy Library: Shell32.lib
 - Add semi-colon followed by “Shell32.lib” to new Additional Dependencies → **Apply** → **Ok**
 - Rebuild and check that this error has gone away – Build Succeeded!
- CLK > F5 and verify two windows: console and gui “Hello World” – Success!

▪ How to Draw a Triangle on the black Hello World Screen

– Using Legacy OpenGL for quick test & debugging code

- Under glClear(GL_COLOR_BUFFER_BIT); add this code and CLK > F5 to Run

```
/* JT Added Legacy OpenGL quick test & debug code here - using 2D projection vertices */
glBegin(GL_TRIANGLES);
glVertex2f(-0.5f, -0.5f);
glVertex2f( 0.0f,  0.5f);
glVertex2f( 0.5f, -0.5f);
glEnd();
```

- Note: processor memory usage keeps climbing
- Using Modern OpenGL for new application
 - Next Video: Using Modern OpenGL in C++

Vid#3 – Using Modern OpenGL in C++

- **Rendering APIs**
 - Windows DirectX / Direct3D
 - Legacy OpenGL 1.1
 - Where to get into the drivers?
 - Get the function declarations and link against them
 - Modern OpenGL
 - Use OpenGL GLEW Extension Wrangler Library that provides the OpenGL Specification's functions, symbols, and constants declarations in a Header file to link to our application(s)
 - The .c implementation file identifies your platform's graphics hardware/driver, finds the appropriate .dll file, and loads all of the corresponding function pointers which access the compiled binary functions that already exist on your platform.
 - Optional GLAD Library (extensions) – more complex than GLEW but with more control
 - Google: The OpenGL Extension Wrangler Library (glew.sourceforge.net)
 - CLK > Binaries Windows 32-bit or 64-bit link > extract "glew-2.1.0-win32.zip" Open File > copy & paste "glew-2.1.0" folder into OpenGL project Dependencies folder
 - Configure Properties > C++ General: Link GLEW include folder in OpenGL project
 - Add GLEW include folder to project
 - Additional Include Directories: \$(SolutionDir)Dependencies\glew-2.1.0\include → **Apply**
 - Configure Properties > Linker General: Link GLEW include folder in OpenGL project
 - Add GLEW library folder to project
 - Additional Library Directories: \$(SolutionDir)Dependencies\glew-2.1.0\lib\Release\Win32 → **Apply**
 - Configure Properties > Linker > Input
 - Additional Dependencies
 - Add "glew32s.lib" to new Additional Dependencies → **Apply** → **Ok**
 - Add "#include <GL/glew.h>" to OpenGL project's "Application.cpp" file
 - Rebuild → C:\Dev\Cherno\OpenGL\Dependencies\glew-2.1.0\include\GL\glew.h(85,1): fatal error C1189: #error: gl.h included before glew.h 1>Done building project "OpenGL.vcxproj" -- FAILED.
 - Solution: move "#include <GL/glew.h>" before ANY other #includes

- Rebuild → error LNK2019: unresolved external symbol __imp__glewInit@0 referenced in function _main. 1>C:\Dev\Cherno\OpenGL\bin\Win32\Debug\OpenGL.exe : fatal error LNK1120: 1 unresolved externals. 1>Done building project "OpenGL.vcxproj" -- FAILED.
- Configure Properties > C++ > Preprocessor
 - Preprocessor Definitions
 - Add "GLEW_STATIC;"<different options> → **Apply** → **Ok**
- Review doc folder's function(s) documentation – open index.html (local HTML version of GLEW documentation)
 - CLK > Usage – Read Documentation on how to initialize GLEW
 - Two important GLEW Issues:
 - First you need to create a valid OpenGL rendering context and call glewInit() to initialize the extension entry points. If glewInit() returns GLEW_OK, the initialization succeeded and you can use the available extensions as well as core OpenGL functionality. Example code:

```
#include <GL/glew.h>
#include <GL/glut.h>
...
glutInit(&argc, argv);
glutCreateWindow("GLEW Test");
GLenum err = glewInit();
if (GLEW_OK != err)
{
    /* Problem: glewInit failed, something is seriously wrong. */
    fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
    ...
}
fprintf(stdout, "Status: Using GLEW %s\n", glewGetString(GLEW_VERSION));
```

- CLK > Building – Read Documentation
 - CLK > Installation – Read Documentation
- Add GLEW initialization code AFTER glfwMakeContextCurrent(window) code


```
/** Make the window's context current - this MUST BE PERFORMED BEFORE glewInit() !!! */
glfwMakeContextCurrent(window);

/** JT Added Modern OpenGL code here - MUST FOLLOW glfwMakeContextCurrent(window) */
if (glewInit() != GLEW_OK)
    std::cout << "glewInit() Error!" << std::endl;
```

 - Set breakpoint on: if (glewInit() != GLEW_OK) → CLK F5 > CLK F10 and note: no errors (GLEW_OK) and now have access to ALL GLEW functions.

- Solutions Folder > OpenGL > External Dependencies > RT+ CLK > glew.h > Open → shows 23,000+ lines of GLEW header function declaration code.

- Defines GLEW function pointers for ALL GLEW functions used in our application

- Example: add glGenBuffers() code and find function definition in glew.h to determine usage requirements

```
unsigned int a;  
glGenBuffers(1, &a);
```

- Find glGenBuffers Macro in glew.h (1709) #define glGenBuffers GLEW_GET_FUN(__glewGenBuffers)

- > RT+CLK on function signature __glewGenBuffers > Go to Definition (F12) > (19967) GLEW_FUN_EXPORT PFNGLGENBUFFERSPROC __glewGenBuffers;

- > RT+CLK on type of function pointer PFNGLGENBUFFERSPROC > Go to Definition (F12) > (1689) typedef void (GLAPIENTRY * PFNGLGENBUFFERSPROC) (GLsizei n, GLuint* buffers);

- Return type: void

- Parameters: GLsizei n, GLuint* buffers – a size and an unsigned int

- Print OpenGL version AFTER we have a valid version

- Example: replace the prior glGenBuffers() code example with this code → F5

```
/** JT Added Print Modern OpenGL Version code here **/  
std::cout << glGetString(GL_VERSION) << std::endl;
```

- Check console window for printed version: 2.1.0 - Build 8.15.10.2900 (Intel driver version)

-

Vid#4 – Vertex Buffers and Drawing a Triangle in OpenGL

- **Modern OpenGL requires creation of a:**
 - **Vertex Buffer**
 - A memory buffer comprised of an array of bytes of GPU (graphics processor unit) memory that an application can push bytes of data into this OpenGL array in graphics video ram.
 - Then issue Draw Call(s) that executes a graphics function which pops bytes of data from this OpenGL array to draw an image on the screen.
 - **Shader**
 - A program that runs on the GPU (graphics processor unit) that defines How the data in the Vertex Buffer is to be drawn.
 - We need to describe how to read and interpret the array bytes of data and how to draw (rasterize) that data onto our screen.
 - **OpenGL runs as a State Machine**
 - Data is NOT to be treated as an object
 - Setup a series of states where each state knows what (drawing shape), where (screen location) and how (sequence of vertices contained in the Vertex Buffer) it's expected to draw the shape.
 - Each state selects a Vertex Buffer and a matching Shader, then executes code to draw a shape (e.g. a triangle)
 - **Creating the Vertex Buffer using the original Legacy OpenGL (3) vertices**
 - Review Legacy OpenGL code (inside while loop) that draws a Triangle shape using (3) vertices with glBegin and glEnd
 - Modern OpenGL – put the (3) vertices data into a Vertex Buffer, send the buffer to OpenGL's video ram, and later issue a Draw Pull requesting the GPU draws what's in the Vertex Buffer.
 - Every OpenGL glGenBuffers requires a unique ID of your object as its 1st parameter and a pointer to the memory address of the unsigned int buffer as its 2nd parameter.
 - create float array of [6] vertices - positions by Alt+Shift Legacy vertices --> Ctrl+c

```
float positions[6] = {  
    -0.5f, -0.5f,  
    0.0f, 0.5f,  
    0.5f, -0.5f  
};
```
 - glGenBuffers(int bufferID, pointer to memory address of unsigned int buffer)
 - unsigned int buffer;
 - glGenBuffers(1, &buffer);
 - Bind or Select Buffer which is the target (type = GL_ARRAY_BUFFER, ID = buffer)
 - glBindBuffer(GL_ARRAY_BUFFER, buffer);
 - Specify the type, size of data to be placed into the buffer using glBufferData
 - glBufferData(GL_ARRAY_BUFFER, 6 * sizeof(float));

- www.docs.gl glBufferData(GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage)
 - glBufferData(GL_ARRAY_BUFFER, 6 * sizeof(float), positions, GL_STATIC_DRAW);
- Two Draw Pull Methods:
 - void glDrawArrays(GLenum mode, GLint first, GLsizei count);
 - mode – specifies what kind of primitive to render (GL_TRIANGLES)
 - first – specifies the starting index in the enabled arrays.
 - count – specifies the number of indices to be rendered
 - glDrawArrays(GL_TRIANGLES, 0, 3);
 - void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices);
 - mode – specifies what kind of primitive to render (GL_TRIANGLES)
 - count – specifies the number of indices 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 an offset of the first index in the array in the data store of the buffer currently bound to the GL_ELEMENT_ARRAY_BUFFER target.

Vid#5 – Vertex Attributes and Layouts in OpenGL

- Missing Parts from Vid#4
 - Vertex Attributes – uses Vertex Attribute pointer(s) to GPU Memory Layout for each primitive type that to be drawn on the screen.
 - **Vertex / Vertices** – are NOT JUST a position. Each **Indexed** Vertex includes a Set of Attributes that typically includes the x, y (optional z) vertex position(s) data BUT can also include other graphical data (layers, textual coordinates, normals, colors, binormals, tangents, etc.)
 - **Index** – tells which index each Vertex Attribute is located/referenced.
 - **Example: index[0] may reference the Vertex's position, index[1] textual coordinate, index[2] normal, etc. are referenced both by the Vertices and the Shader.**
 - void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer);
 - index – specifies the index of the generic vertex attribute to be modified.
 - size – specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4. Additionally, the symbolic constant GL_BGRA is accepted by glVertexAttribPointer. The initial value is 4.
 - **Example: float positions[6] { -0.5f, -0.5f, 0.0f, 0.5f, 0.5f, -0.5f }; has (3) x, y location pairs at index[0] so set the size = 2 for a two component vector that represents each Vertex position.**
 - type – specifies the data type of each component in the array.
 - **Example: symbolic constant = GL_FLOAT**
 - normalized – for glVertexAttribPointer, specifies whether fixed-point data values should be normalized (GL_TRUE) or converted directly as fixed-point values (GL_FALSE) when they are accessed.

- **Example: converted directly as fixed-point values (GL_FALSE). Note: could be handled by C++**
- **stride** – specifies the byte offset between consecutive generic vertex attributes. If stride is 0, the generic vertex attributes are understood to be tightly packed in the array. The initial value is 0.
 - **Example: the amount of bytes between each Vertex based on a 3 component position vector of 3 floats = 12 bytes, a 2 component vector textual coordinate of 2 floats = 8 bytes, and a 3 component normal vector of 3 floats = 12 bytes where each float is 4 bytes in length for a total stride of 32 bytes for each Vertex.**
 - **OpenGL: the amount of bytes between each Vertex based on a 2 (x, y) component position vector of 2 floats = 8 bytes.**
- **Pointer** – specifies an offset of the first component of the first generic vertex attribute in the array in the data store of the buffer currently bound to the GL_ARRAY_BUFFER target. The initial value is 0
 - **Example: position has an offset pointer = 0, textual coordinate has an offset pointer = 12, and the normal has an offset pointer = 20**
- **glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float) * 2, 0); which requires: glEnableVertexAttribArray(GLuint index);**
- Specify the location and data format of the array of generic vertex attributes at index index to use when rendering. size specifies the number of components per attribute and must be 1, 2, 3, 4, or GL_BGRA. type specifies the data type of each component, and stride specifies the byte stride from one attribute to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.
- For glVertexAttribPointer, if normalized is set to GL_TRUE, it indicates that values stored in an integer format are to be mapped to the range [-1,1] (for signed values) or [0,1] (for unsigned values) when they are accessed and converted to floating point. Otherwise, values will be converted to floats directly without normalization.
- If pointer is not NULL, a non-zero named buffer object must be bound to the GL_ARRAY_BUFFER target (see glBindBuffer), otherwise an error is generated. pointer is treated as a byte offset into the buffer object's data store. The buffer object binding (GL_ARRAY_BUFFER_BINDING) is saved as generic vertex attribute array state (GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING) for index index.
- When a generic vertex attribute array is specified, size, type, normalized, stride, and pointer are saved as vertex array state, in addition to the current vertex array buffer object binding.
- To enable and disable a generic vertex attribute array, call glEnableVertexAttribArray and glDisableVertexAttribArray with index. If enabled, the generic vertex attribute array is used when glDrawArrays, glMultiDrawArrays, glDrawElements, glMultiDrawElements, or glDrawRangeElements is called.
- Each generic vertex attribute array is initially disabled and isn't accessed when glDrawElements, glDrawRangeElements, glDrawArrays, glMultiDrawArrays, or glMultiDrawElements is called.
- Errors:
 - GL_INVALID_VALUE is generated if index is greater than or equal to GL_MAX_VERTEX_ATTRIBS.
 - GL_INVALID_VALUE is generated if size is not 1, 2, 3, 4 or (for glVertexAttribPointer), GL_BGRA.

- GL_INVALID_ENUM is generated if type is not an accepted value.
- GL_INVALID_VALUE is generated if stride is negative.
- GL_INVALID_OPERATION is generated if size is GL_BGRA and type is not GL_UNSIGNED_BYTE, GL_INT_2_10_10_10_REV or GL_UNSIGNED_INT_2_10_10_10_REV.
- GL_INVALID_OPERATION is generated if type is GL_INT_2_10_10_10_REV or GL_UNSIGNED_INT_2_10_10_10_REV and size is not 4 or GL_BGRA.
- GL_INVALID_OPERATION is generated if type is GL_UNSIGNED_INT_10F_11F_11F_REV and size is not 3.
- GL_INVALID_OPERATION is generated by glVertexAttribPointer if size is GL_BGRA and normalized is GL_FALSE.
- GL_INVALID_OPERATION is generated if zero is bound to the GL_ARRAY_BUFFER buffer object binding point and the pointer argument is not NULL.
- Associated Gets
 - glGet with argument GL_MAX_VERTEX_ATTRIBS
 - glGetVertexAttrib with arguments index and GL_VERTEX_ATTRIB_ARRAY_ENABLED
 - glGetVertexAttrib with arguments index and GL_VERTEX_ATTRIB_ARRAY_SIZE
 - glGetVertexAttrib with arguments index and GL_VERTEX_ATTRIB_ARRAY_TYPE
 - glGetVertexAttrib with arguments index and GL_VERTEX_ATTRIB_ARRAY_NORMALIZED
 - glGetVertexAttrib with arguments index and GL_VERTEX_ATTRIB_ARRAY_STRIDE
 -
 - glGetVertexAttrib with arguments index and GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING
 - glGet with argument GL_ARRAY_BUFFER_BINDING
 - glGetVertexAttribPointerv with arguments index and GL_VERTEX_ATTRIB_ARRAY_POINTER
- Examples
 - **void glEnableVertexAttribArray(GLuint index);**
 - **index** – specifies the index of the generic vertex attribute to be enabled or disabled.
 - **glEnableVertexAttribArray(0);**
 - Run & Test – F5: Shows same Triangle as the original Legacy OpenGL code without ANY Shader.
 - Why? Some GPU Drivers provide a Default Shader (Dell Latitude E6410 does) if you HAVE NOT specified your own Shader.
- Shaders – must match the GPU Vertex Attributes Layout on the CPU C++ side

Vid#6: How Shaders Work in OpenGL

- **Run & Test – F5:** Shows same Triangle as the original Legacy OpenGL code without ANY Shader.
 - Why? Some GPU Drivers provide a Default Shader (Dell Latitude E6410 implements default pixel color = white) if you HAVE NOT specified your own Shader.
- **Build our own Custom Shader** – a program, block of text code that is compiled, linked and executes/runs, on your GPU (Graphical Processing Unit)
- **Two Most Popular Shader Types:**
 - **Vertex Shaders**
 - Code gets called for each Vertex of a primitive we are trying to render (e.g. 3 vertices Triangle gets called 3 times, once for each vertex).
 - The primary function of a Vertex Shader is to tell OpenGL where you want that Triangle vertex positions to be on your virtual screen space.
 - It also passes attribute(s) data into the next stage (Fragment Shader).
 - The attribute(s) data can be accessed by the Vertex Shader code using the index parameter.
 - **Fragment (Pixel) Shaders**
 - Difference between Fragments and Pixels?
 - Code runs once for each pixel in the primitive (shape) that needs to get Rasterized (primitive filled in on screen's window).
 - The primary function of a Fragment (Pixel) Shader is to decide what color each pixel is drawn.
 - Typical examples of code that runs in the Fragment Shader are: Lighting (each pixel has a color value that is determined by the lighting value, texture value, material value, camera position, environment properties)
- **OpenGL Graphics Rendering Pipeline**
 - Write graphics data on the CPU, bound certain states
 - CPU issues a Draw Call to the GPU that first calls a Vertex Shader and then the Fragment (Pixel) Shader gets called.
 - GPU executes the Shader code triggered by the Draw Call and draws/rasterizes the graphic pixels on the screen
- **OpenGL Shader code requirements:**
 - Use `glEnable(GL_VERTEX_SHADER)` to enable the Shader
 - Shaders work based on the State Machine
- **OpenGL Uniform**
 - Send data from the CPU to the GPU using:
 - `void glUniform1f(GLint location, GLfloat v0);`

Vid#7: Writing a Shader in OpenGL

- Create Shader static int function before main()
 - create static int CreateShader function with parameters:
 - const string pointer vertexShader(actual source code),
 - const string pointer fragmentShader (actual source code)
 - Many ways to Create and Compile Shaders
 - Read in as C++ string that contains the Shader source code (method used in this simple example)
 - Read in from a file
 - Download from internet
 - Read in as binary data
 - Provide OpenGL with our source code (two Shader strings) and want OpenGL to compile that program linking the vertexShader and fragmentShader together into a single Shader program returning a unique int identifier back to this program in order to bind this Shader and use it in our application.
 - void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)
 - void glAttachShader(GLuint program, GLuint shader);
 - void glLinkProgram(GLuint program);
 - void glValidateProgram(GLuint program);
 - void glDeleteShader(GLuint shader);
 - void glGetShaderiv(GLuint shader, GLenum pname, GLint *params);
 - void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infoLog);
 - Write a Shader and Test it
 - Create vertexShader
 - using #version 330 - NOT the latest version b/c we don't need all those features yet
 - core - DOES NOT ALLOW ANY Deprecated functions
 - "layout(location = id is 1st param of glVertexAttribPointer) in vec4(2nd param) position;"

```
std::string vertexShader =  
"#version 330 core \n"  
"\n"  
"layout(location = 0) in vec4 position;"  
"\n"  
"void main()\n"  
"{\n"  
"  gl_Position = position\n"  
"}\n";
```

- **Create fragmentShader**
 - using #version 330 - NOT the latest version b/c we don't need all those features yet
 - core - DOES NOT ALLOW ANY Deprecated functions
 - "layout(location = id is 1st param of glVertexAttribPointer) out vec4(2nd param) color;"

```
std::string fragmentShader =
"#version 330 core \n"
"\n"
"layout(location = 0) out vec4 color;\n"
"\n"
"void main()\n"
"{\n"
"    color = vec4(1.0, 0.0, 0.0, 1.0);\n"
"}\n";
```

- **GLuint glCreateShader(GLenum shaderType);**
 - /* Call to create vertexShader and fragmentShader above */
 - unsigned int shader = CreateShader(vertexShader, fragmentShader);
- **void glUseProgram(GLuint program);**
 - /* Bind our Shader */
 - glUseProgram(shader);
- **Run F5 – builds a white (NOT a red) Triangle**
 -

Vid#8: How I Deal with Shaders in OpenGL

▪ Concepts:

- Convert Vid#7 vertexShader and fragmentShader from two strings in the main() code that requires “\n” at the end of each line to a external file that contains both Shaders separated by two string codes. This behaves like DirectX GPU Shader API.
 - Alternative is to create (2) separate Shader files, one for vertexShader and one for fragmentShader.
- Create a file “ ” that contains the original two vertexShader and fragmentShader
 - RT+CLK > OpenGL > Add new folder > “res” > Add
 - RT+CLK > res > Add new folder > “shaders” > Add
 - RT+CLK > shader > Add new item > “Basic.shader” > Add
- Copy & Paste the two original vertexShader and fragmentShader code in the Application.cpp file into the new Basic.shader file and modify the two Shaders as follows:
 - Delete “std::string vertexShader = “ line
 - Delete “std::string fragmentShader = “ line
 - Ctrl + h > “ > Replace all (Alt + a) > Ok
 - Ctrl + h > \n > Replace all (Alt + a) > Ok
 - Add “#shader vertex” at top of vertexShader
 - Add “#shader fragment” at top of fragmentShader
- Add new function ParseShader to parse external Basic.shader file
 - #include <fstream>

```
/** Vid#8 Add new function ParseShader to parse external Basic.shader file
returns - struct ShaderProgramSource above which contains two strings (variables)
note: C++ functions are normally capable of only returning one variable ***/
static ShaderProgramSource ParseShader(const std::string& filepath)
{
    /* open file */
    std::ifstream stream(filepath);

    /* create enum class for each Shader type */
    enum class ShaderType
    {
        NONE = -1, VERTEX = 0, FRAGMENT = 1
    };
};
```



```

/* parse file line by line */
std::string line;

/* define buffers for 2 Shaders: vertexShader and fragmentShader */
std::stringstream ss[2];

/* set initial ShaderType = NONE */
ShaderType type = ShaderType::NONE;

while (getline(stream, line))
{
    /* find "#shader" keyword */
    if (line.find("#shader") != std::string::npos)
    {
        if (line.find("vertex") != std::string::npos)
            /* set mode to vertex */
            type = ShaderType::VERTEX;

        else if (line.find("fragment") != std::string::npos)
            /* set mode to fragment */
            type = ShaderType::FRAGMENT;
    }
    else
        /* add each line to the corresponding buffer after detecting the ShaderType */
        {
            /* type is an index to push data into the selected array buffer, casted to a Shader int type,
            to add each new line plus newline char */
            ss[(int)type] << line << '\n';
        }
}

```

```
/* returns a struct comprised of two ss strings */  
return { ss[0].str(), ss[1].str() };  
}
```

- Create struct for returning multiple C++ variables

```
/** Create a struct that allows returning multiple items **/  
struct ShaderProgramSource  
{  
    std::string VertexSource;  
    std::string FragmentSource;  
};
```

- Modifications to main() function: (per LearnOpenGL Hello-Triangle webpage)

- Re-specified positions[] array as 3 x 3 (x, y, and z)

```
float positions[] = {  
    -0.5f, -0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    0.0f, 0.5f, 0.0f  
};
```

- changed 2nd param: `glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW);`
- changed 2nd and 5th params: `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0);`
- changed vertexShader:

```
#shader vertex  
#version 330 core  
layout(location = 0) in vec4 position;  
void main()  
{  
    gl_Position = position;  
};
```

- changed fragmentShader:

```
#shader fragment  
#version 330 core
```

```
out vec4 FragColor;

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

- Test new code: (F5)
 - OpenGL Properties page > Configuration Properties > Debugging > \$(ProjectDir) – default directory > **Ok**
 - Chernov's Shader code DOES NOT COMPILE! IT FAILS because of OpenGL Ver. 3.3 layout syntax in vertexShader and color syntax in fragmentShader:
 - Replaced Chernov vertexShader code: with LearnOpenGL Hello-Triangle code
 - Replace Chernov fragmentShader code: with LearnOpenGL Hello-Triangle code
 - **Results: Correctly produces a Red Triangle**

Vid#9: Index Buffers in OpenGL

▪ Concepts:

- Every primitive is based on Triangles?
 - Drawing a Square is implemented with right (2) Triangles that share (2) vertices positions

- Modify positions[] array C++ code:

```
/* Vid#9: add 2nd set of (3) x, y, z vertex positions for 2nd inverted triangle added
to original right triangle forming a new Rectangle */
/* Vid#8: modified to (3) x, y, and z vertex positions per LearnOpenGL */
/* Vid#4: JT Define Vertex Buffer code based on Vid#2 example commented out below */
/* create float array of [3] Vertices - (3) x, y, z vertex position pairs by Alt+Shift Legacy vertices --> Ctrl + c */
float positions[] = {
    -0.5f, -0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
    0.5f, 0.5f, 0.0f,
    0.5f, 0.5f, 0.0f,
    -0.5f, 0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f
};
```

- Modify glDrawArrays(GL_TRIANGLES, 0, 6) code:

```
/* Vid#9: modify buffer = 3 to buffer = 6 for (2) adjacent Triangles forming a Rectangle */
/* Vid#4: Modern OpenGL draws what's in the new Vertex Buffer */
/* glDrawArrays(GL_TRIANGLES, 0, 3); draws a Triangle based on the last glBindBuffer(GL_ARRAY_BUFFER, buffer); */
glDrawArrays(GL_TRIANGLES, 0, 6);
```

- Result: generated a window with a blue rectangle comprised of two triangles sharing two vertices.
- Note: this is a sub-optimal method of creating a Rectangle from two Triangles

- Use an Index Buffer using existing Vertices (4:20)

- Vid#9B: remove 2 duplicate vertices of the 6 vertices in position[] to implement an Index Buffer */

```
/* redefine positions array with 4 vertices to create 2 triangles
```

```
float positions[] = {
    -0.5f, -0.5f, 0.0f, // vertex 0
    0.5f, -0.5f, 0.0f, // vertex 1
    0.5f, 0.5f, 0.0f, // vertex 2
    -0.5f, 0.5f, 0.0f, // vertex 3
};
```

- Create new Index Buffer using unsigned int indices[] array for drawing the two triangles to form a rectangle

```
/* Vid9B: create Index Buffer using new indices[] array
   note: must be unsigned but can use char, short, int, etc. */
unsigned int indices[] = {
    0, 1, 2,    // 1st right triangle
    2, 3, 0    // 2nd inverted right triangle
};
```

- Copy Vid5: original glGenBuffers(), glBindBuffer(), and glBufferData() calls and Create new Index Buffer calls

```
/* Vid9: new Index Buffer calls */
/* glGenBuffer(int bufferID, pointer to memory address of unsigned int buffer) creates buffer and provides and ID */
unsigned int ibo;
glGenBuffers(1, &ibo);
/* Bind or Select Buffer which is the target (type = GL_ELEMENT_ARRAY_BUFFER, ID = ibo) */
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
/* Specify the type, size of data to be placed into the buffer */
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

- Replace glDrawArrays(GL_TRIANGLES, 0, 6); with glDrawElements()

```
/** Vid#9 new Draw call to glDrawElements(GL_TRIANGLES, #indices, type, ptr to index buffer
    or nullptr b/c we bound it using glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); ) */
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr);
```

- Test new code: (F5)

- **Results (12:05) : Correctly produces a Blue Rectangle**
- Note: changing glDrawElements(GL_TRIANGLES, 6, GL_INT, nullptr); from specifying unsigned int to int will draw a black screen with no primitive(s) drawn BECAUSE ALL Index Buffers MUST be specified AS unsigned int!

- Need to create a way to troubleshoot, diagnose, and debug OpenGL programming errors in C++ and Shader code

Vid#10: Dealing with Errors in OpenGL

- **Concepts: What does OpenGL provide to decipher errors?**
 - **glGetError** (compatible with all OpenGL versions 2.0 through 4.5 and returns error flag(s) that indicate the type of error that occurred.
 - Each call to **glGetError()** returns (1) flag in the order of occurrence.
 - **GLenum glGetError(void);**
 - The following errors are currently defined:
 - **GL_NO_ERROR** – No error has been recorded. The value of this symbolic constant is guaranteed to be 0.
 - **GL_INVALID_ENUM** – An unacceptable value is specified for an enumerated argument. The offending command is ignored and has no other side effect than to set the error flag.
 - **GL_INVALID_VALUE** – A numeric argument is out of range. The offending command is ignored and has no other side effect than to set the error flag.
 - **GL_INVALID_OPERATION** – The specified operation is not allowed in the current state. The offending command is ignored and has no other side effect than to set the error flag.
 - **GL_INVALID_FRAMEBUFFER_OPERATION** – The framebuffer object is not complete. The offending command is ignored and has no other side effect than to set the error flag.
 - **GL_OUT_OF_MEMORY** – There is not enough memory left to execute the command. The state of the GL is undefined, except for the state of the error flags, after this error is recorded.
 - **GL_STACK_UNDERFLOW** – An attempt has been made to perform an operation that would cause an internal stack to underflow.
 - **GL_STACK_OVERFLOW** – An attempt has been made to perform an operation that would cause an internal stack to overflow.
 - **glGetError Description:**
 - **glGetError** returns the value of the error flag. Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until **glGetError** is called, the error code is returned, and the flag is reset to **GL_NO_ERROR**. If a call to **glGetError** returns **GL_NO_ERROR**, there has been no detectable error since the last call to **glGetError**, or since the GL was initialized.
 - To allow for distributed implementations, there may be several error flags. If any single error flag has recorded an error, the value of that flag is returned and that flag is reset to **GL_NO_ERROR** when **glGetError** is called. If more than one flag has recorded an error, **glGetError** returns and clears an arbitrary error flag value. Thus, **glGetError** should always be called in a loop, until it returns **GL_NO_ERROR**, if all error flags are to be reset.
 - Initially, all error flags are set to **GL_NO_ERROR**.
- **OpenGL Test-01 (change GL_UNSIGNED_INT to GL_INT)**
 - Call **glGetError()** in a while loop until ALL OpenGL errors have been cleared
 - Call **glDrawElements(GL_TRIANGLES, 6, GL_INT, nullptr);**
 - **glGetError()** in a while loop to display ALL OpenGL errors from the **glDrawElements(GL_TRIANGLES, 6, GL_INT, nullptr)** Call
- In OpenGL 4.3 (4:30) **glDebugMessageCallback()**

- `void glDebugMessageCallback(DEBUGPROC callback, void * userParam);`
 - callback – Specifies the address of a callback function (*ptr) when an error occurs that will be called when a debug message is generated.
 - userParam – A user supplied pointer that will be passed on each invocation of callback.
- The callback function should have the following 'C' compatible prototype:
 - `typedef void (APIENTRY *DEBUGPROC)(GLenum source, GLenum type, GLuint id, GLenum severity, GLsizei length, const GLchar *message, const void *userParam);`
- Unlike `glGetError()` having to be called many times for multiple errors, `glDebugMessageCallback()` is called only Once
 - And, it contains plain English explanation of each error and suggestions on how to fix each error.
- Vid#10 (9:00): Test `glGetError()` only in this tutorial
 - 1st Call `GLClearErrors();`

```
/* Vid10: add new GLClearErrors() static function that returns void */
static void GLClearErrors()
{
    /* loop while there are errors and until GL_NO_ERROR is returned */

    while (glGetError != GL_NO_ERROR);
}
```
 - 2nd Call `glDrawElements(GL_TRIANGLES, 6, GL_INT, nullptr);`
 - 3rd Call `GLCheckErrors();`

```
/* Vid10: add new GLCheckErrors() static function that returns unsigned enum (int) in order */
static void GLCheckErrors()
{
    while (GLenum error = glGetError())
    {
        std::cout << "[OpenGL Error] (" << error << ")" << std::endl;
    }
}
```
- Vid#10 (13:30): Using an Assertion break in our code
 - `/* Vid#10: (14:30) add ASSERT(x) macro to validate a condition and call a breakpoint if true using the MSVC function __debugbreak() */`
 - `#define ASSERT(x) if (!(x)) __debugbreak();`
- `/* Vid#10: (16:20) GLCall(x) macro where (x) is the call function to Clear OpenGL Error(s) that calls the GLClearErrors() function */`
- `/* Vid#10 (18:45) use macros to find out which line of code this errored function occurred.`

- In GLLogCall(x) - changed to a string (#x) for printing the file name (__FILE__), and printing the line number (__LINE__) */

```
#define GLCall(x) GLClearErrors();\  
x;\  
ASSERT(GLLogCall(#x, __FILE__, __LINE__))
```

- /* Vid#10: (20:30) wrap GLCall() around these (3) gl calls */

```
unsigned int buffer;  
GLCall(glGenBuffers(1, &buffer));  
  
/* Bind or Select Buffer which is the target (type = GL_ARRAY_BUFFER, ID = buffer) */  
GLCall(glBindBuffer(GL_ARRAY_BUFFER, buffer));  
  
/* Specify the type, size of data to be placed into the buffer */  
GLCall(glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW));
```

- /* Vid#10: (20:30) wrap GLCall() around these (3) gl calls */

```
unsigned int ibo;  
GLCall(glGenBuffers(1, &ibo));  
  
/* Bind or Select Buffer which is the target (type = GL_ELEMENT_ARRAY_BUFFER, ID = ibo) */  
GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo));  
  
/* Specify the type, size of data to be placed into the buffer */  
GLCall(glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW));
```

- /* Vid#10 (17:30) replace this code with the 2nd ASSERT GLCall wrapped around the glDrawElements() call */

```
//GLClearErrors();  
//glDrawElements(GL_TRIANGLES, 6, GL_INT, nullptr);  
//ASSERT(GLLogCall());  
  
GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr));
```

- The End

Vid#11: Uniforms in OpenGL

▪ What are Uniforms and Where are they used in OpenGL?

- A way of getting C++ Data into a GPU OpenGL Shader used like a variable
- Vid#10 hardcoded the Blue FragColor = vec4(0.0f, 0.0f, 1.0f, 1.0f); in the FragmentShader.
- Goal – is to define the Blue FragColor in the C++ code and pass it into the Shader and update it as necessary
 - Two Methods:
 - Vertex Buffer Attributes – are Setup per Vertex
 - Uniforms – are Setup per Draw BEFORE the GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr));
- Focus – this Vid#11 (3:45) will focus on Uniforms for passing in C++ Data to a Shader
 - GLCall(int location = glGetUniformLocation(shader, "u_Color")); // Retrieve location of "u_Color" color variable
 - ASSERT(location != -1); // means we could NOT find our uniform, WAS Found but NOT used, Or Errored when called.
 - void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3); // Call glUniform4f with location and 4 floats representing the Blue Triangle color

```
/* Vid#11 (3:45) glUniform — Specify the value of a uniform (vec4 = 4 floats) variable for the current program object */  
/* void glUniform4f(GLint location,
```

```
    GLfloat v0,  
    GLfloat v1,  
    GLfloat v2,  
    GLfloat v3);
```

each uniform gets assigned a unique ID name for referencing

```
GLCall(int location = glGetUniformLocation(shader, "u_Color"));  
ASSERT(location != -1); // means we could NOT find our uniform, WAS Found but NOT used, Or Errored when called.  
GLCall(glUniform4f(location, 0.2f, 0.3f, 0.0f, 1.0f)); is used to pass in the Blue Triangle "u_Color"  
into the Fragment Shader in Basic.shader */
```

```
/* Vid#11: (7:00) : Retrieve the int location of the variable location */  
GLCall(int location = glGetUniformLocation(shader, "u_Color"));
```

```
/* Check if location DID NOT return -1 */  
ASSERT(location != -1);
```

```
/* Determine the type of Data to send to the GPU's Shader - (4) floats 1st param - int location of these (4) floats */  
GLCall(glUniform4f(location, 0.8f, 0.3f, 0.8f, 1.0f));
```

- Run & Test (F5): Passed! – correctly produced the Blue Triangle based on the glUniform4f() call that passed in the (4) floats to the fragmentShader in the new Basic.shader

– Animate/Change the Triangle's color over time (8:00)

- /* Vid#11 (3:40) Bind our Shader - now wrapped in GLCall() to check if shader was found */

```
GLCall(glUseProgram(shader));
```

- /* Vid#11 (8:15) define 4 float variables: r, g, b, and i */

```
float r = 0.0f; // red color float var initially set to zero
```

```
float increment = 0.05f; // color animation float increment var initially set to 0.05
```

- /* Vid#11 (8:15) copy & paste glUniform4f() GLCall here & wrap GLCall around glUniform4f() and glDrawElements() calls */

- **Note: glUniform's CANNOT be changed between drawing ANY elements contained within a glDraw call**

- **Which means you can't draw one triangle in one color and the other triangle in another color**

```
GLCall(glUniform4f(location, r, 0.3f, 0.8f, 1.0f)); // note: (1) Uniform required BEFORE/PER each glDraw
```

```
GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr));
```

- /* Vid#11 (8:30) check if r value > 1.0f --> set increment = -0.05f, else if r value < 0.0f --> set increment = 0.05f */

```
if (r > 1.0f)
```

```
    increment = -0.05f;
```

```
else if (r < 0.0f)
```

```
    increment = 0.05f;
```

```
r += increment;
```

- /* Vid#11 (9:00) - should sync our Swap with the monitor's refresh rate and produce a smooth color change transition */

```
glfwSwapInterval(1);
```

- Run & Test (F5) (9:00) : Passed! The Triangle changes from Pink to Blue at monitor refresh rate intervals.

- The End

Vid#12: Vertex Arrays in OpenGL

▪ What are the differences between Vertex Buffers and Vertex Arrays?

- OpenGL is the only GPU API that provides for Vertex Arrays.
- Vertex Arrays are a way of binding Vertex Buffers with a specification for a specific layout.
- Vertex Array objects allow us to bind our Vertex Buffer specification by using an `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0);` to an actual Vertex Buffer or a series of Vertex Buffers.
 - This works Ok if we our binding one fragmentShader, one Vertex Buffer and one Index Buffer for one object to execute one `glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr)` command. The downside is we would have to re-bind everything for each graphical object to draw.
 - Instead of specifying a Vertex Buffer Layout and Index Buffer every time we execute a `glDraw...` command:
 - Unbind the Shader (shader), the vertex buffer (buffer), and the index buffer (ibo) by setting each = 0 and re-bind all (3) inside the Rendering while loop before the `glDraw` command.

```
/* Vid#12: (4:00) Unbind the Shader (shader), the vertex buffer (buffer), and the index buffer (ibo)
   by setting each = 0 and re-bind all (3) inside the Rendering while loop before the glDraw cmd */
GLCall(glUseProgram(0));
GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0));
```

- And inside the Rendering while loop, Bind Shader (shader), Uniform (location), Vertex Buffer (buffer) and Index Buffer (ibo) BEFORE calling `glDrawElements...`

```
/* Vid#12: (4:45) Bind Shader (shader), Uniform (location), Vertex Buffer (buffer)
   and Index Buffer (ibo) BEFORE calling glDrawElements... */

GLCall(glUseProgram(shader));           // bind our shader
GLCall(glUniform4f(location, r, 0.3f, 0.8f, 1.0f)); // setup uniforms

GLCall(glBindBuffer(GL_ARRAY_BUFFER, buffer)); // bind our Vertex Array Buffer (buffer)
GLCall(glEnableVertexAttribArray(0));         // enable vertex attributes of index 0
GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0)); // set vertex attributes

GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo)); // bind our Index Buffer (ibo)

GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr)); // Draw Elements call
```

- Run & Test (F5): (5:30) Passed! Draws Triangle that changes colors from pink to blue and back to pink at monitor refresh rate.
- Make a Vertex Array object for each Vertex Buffer Layout
 - Vertex Array objects contain the state of of Vertex Buffers, Index Buffers, and Shaders.

- If we make a Vertex Array object for drawing each piece of geometry, we could theoretically just bind the Vertex Array object(s) because it would bind a Vertex Buffer(s) and their respective Vertex specification layout(s).
- Old Binding process (6:35):
 - Bind our Shader
 - Bind our Vertex Buffer
 - Setup the Vertex Layout
 - Setup and Bind our Index Buffer
 - Issue the glDraw call
- New Binding process (6:50):
 - Bind our Shader
 - Bind our Vertex Array (equivalent to binding Vertex Buffer and Setting up its Layout that contains all of the needed State elements)
 - Bind our Index Buffer
 - Issue the glDraw call
- Vertex Array objects are mandatory (7:00)
 - The OpenGL Compatibility Profile automatically creates a Vertex Array object by default and are MANDATORY. While the Call Profile DOES NOT. Run & Test (F5) (8:00) Passes.
- Tell OpenGL GLFW to create an Open Context and Window with the Core Profile (8:10)
 - Add this code:

```
/* Vid#12 (8:10) OpenGL GLFW Version 3.3 create an Open Context and Window with the Core Profile */
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_COMPAT_PROFILE);
```

- Run & Test (F5) Failed! Console output produced:

```
C:\Dev\Cherno\OpenGL\bin\Win32\Debug\OpenGL.exe (process 9760) exited with code -1.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically
close the console when debugging stops.
Press any key to close this window . . .
```

- Commenting out these (3) lines of code produces the correct Pink to Blue Triangle. And, the console output produces:

```
2.1.0 - Build 8.15.10.2900
```

- A newly-created VAO has array access disabled for all attributes. Array access is enabled by binding the VAO in question and calling:
 - `void glEnableVertexAttribArray(GLuint index);`
 - `glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);`
 - (OpenGL Error) 1282: `GL_INVALID_OPERATION` is generated by `glEnableVertexAttribArray` and `glDisableVertexAttribArray` if no vertex array object is bound.
 - Solution: In CORE Profile, Create Vertex Array object
 - `/* Vid#12: (10:00) Create Vertex Array Object (vao) BEFORE Creating Vertex Buffer (buffer) */`

```
/* Vid#12: (10:00) Create Vertex Array Object (vao) BEFORE Creating Vertex Buffer (buffer)
Create unsigned int Vertex Array Object ID: vao
GLCall(glGenVertexArrays(numVAs, stored vao IDrefptr))
GLCall(glBindVertexArray(VAO ID)) */

unsigned int vao;
GLCall(glGenVertexArrays(1, &vao));
GLCall(glBindVertexArray(vao));
```
 - Run & Test (F5) – Passed BUT required changing Major (2) and Minor (1) Versions to match default in order to work. Any other combination of int versions fail (no window screen). Root Cause: unknown


```
/* Vid#12 (8:10) OpenGL GLFW Version 3.3 create an Open Context and Window with the Core
Profile GLFW_OPENGL_CORE_PROFILE */

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
//glfwWindowHint(GLFW_OPENGL_ANY_PROFILE, GLFW_OPENGL_COMPAT_PROFILE);
glfwWindowHint(GLFW_OPENGL_ANY_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```
 - There is a similar `glDisableVertexAttribArray` function to disable an enabled array.
 - Remember: all of the state below is part of the VAO's state, except where it is explicitly stated that it is not. A VAO must be bound when calling any of those functions, and any changes caused by these function will be captured by the VAO.
 - The compatibility (`glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_COMPAT_PROFILE);`) OpenGL profile makes VAO object 0 a default object.
 - The core OpenGL profile makes VAO object 0 not an object at all. So, if VAO 0 is bound in the core profile, you should not call any function that modifies VAO state. This includes binding the 'GL_ELEMENT_ARRAY_BUFFER' with `glBindBuffer`.

- Vid#12: (11:00) comment out (3) GLCalls to `glBindBuffer(GL_ARRAY_BUFFER, buffer)`, `glEnableVertexAttribArray(0)`, and `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0)`

```
//GLCall(glBindBuffer(GL_ARRAY_BUFFER, buffer));    // bind our Vertex Array Buffer (buffer)
//GLCall(glEnableVertexAttribArray(0));            // enable vertex attributes of index 0
//GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0)); // set vertex attributes
```

- /* Vid#12: (4:00) Unbind the Shader (shader), the vertex buffer (buffer), and the index buffer (ibo)
- by setting each = 0 and re-bind all (3) inside the Rendering while loop before the `glDraw` cmd */
- /* Vid#12 (11:10) add clear `glBindVertexArray(0)` binding */

```
GLCall(glBindVertexArray(0));
GLCall(glUseProgram(0));
GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0));
```

- /* Vid#12: (11:15) add binding `GLCall(glBindVertexArray(vao))` and then bind Index Array Buffer (ibo)

```
GLCall(glBindVertexArray(vao));    // bind our Vertex Array Object
GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo)); // bind our Index Buffer (ibo)
```

- Run & Test (F5) - this works b/c we are linking our Vertex Buffer to our Vertex Array Object */
- When we bind a Vertex Array and bind a Buffer, nothing actually links the two.
- However when we specify the `GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0));`, **index 0 (1st param)** of this Vertex Array is going to be bound to the currently bound `glBindBuffer(GL_ARRAY_BUFFER, buffer)`

```
/* Enable or disable a generic vertex attribute array for index = 0 */
```

```
GLCall(glEnableVertexAttribArray(0));
```

```
/* define an array of generic vertex attribute data
```

index = 0 1st param,

size = 3 2nd param for a (3) component vector that represents each Vertex position,

symbolic constant = `GL_FLOAT` 3rd param,

normalized = converted directly as fixed-point values (`GL_FALSE`) 4th param,

stride = the amount of bytes between each Vertex based on

2nd param vec2 (x, y, z) component position vector of 3 floats = 12 bytes,

pointer = position has an offset pointer = 0 */

```
GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0));
```

- Run & Test (F5): (11:40) Passed! Draws Triangle that changes colors from pink to blue and back to pink at monitor refresh rate

- Additional Notes:

- IF we USE the **GLFW_OPENGL_CORE_PROFILE**, WE MUST CREATE A VERTEX ARRAY OBJECT!
- IF we USE the **GLFW_OPENGL_COMPAT_PROFILE**, IT DOES NOT MEAN there are NO Vertex Array Objects, IT DOES MEAN we have a DEFAULT Vertex Array Object that is BOUND and Available to use.
- Option A (1 VAO for Drawing ALL Geometries): Technically, we could create ONE Vertex Array Object and leave it BOUND for the duration of the program. Then we could BIND a Vertex Buffer and SPECIFY a Vertex Layout EVERY TIME we want to DRAW our Vertex Geometry.
- Option B (Multiple VAOs, 1 for EACH Geometry): For EVERY piece of Geometry you want to Draw, you Create a Vertex Array Object, Specify that specification ONCE by Enabling ANY glEnableVertexArray(s) you want, Specify glVertexAttribPointer(s) as MANY TIMES as needed for setup, BIND Vertex Buffer, and then BIND a different Vertex Array Object EVERY TIME you Draw a different Geometry, then BIND an Index Buffer, and finally Call GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr) OR whatever glDraw function required.
- Which Option A or B is optimal, faster?
 - In the past, Nvidia has a paper (Source Engine) showing Option A was faster discouraging using multiple VAOs
 - This author recommends using Option B with multiple VAOs as NOW recommended by OpenGL as they are currently faster than Option A. However, it may not always be the best option depending on your application.
- Bottom Line: setup a test in your production environment, test both options and log the test results to compare performance.

- IMPORTANT LIMITATION:

```
/* Vid#12 (8:10) OpenGL GLFW Version 3.3 create an Open Context and Window with the Core Profile
GLFW_OPENGL_CORE_PROFILE
```

Note: ONLY (GLFW_CONTEXT_VERSION_MAJOR, 2) and (GLFW_CONTEXT_VERSION_MINOR, 1) WORKS!!!

All other combinations of ints (e.g. 2, 3) of later major/minor versions Fails with console output msg:

C:\Dev\Cherno\OpenGL\bin\Win32\Debug\OpenGL.exe (process 4936) exited with code -1.

To automatically close the console when debugging stops,

enable Tools->Options->Debugging->Automatically close the console when debugging stops.

Press any key to close this window . . . */

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_ANY_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

Vid#13: Abstracting OpenGL into Classes

- **How to Abstract OpenGL code into C++ Classes for cleaner, easier to debug, parameterizable, Re-Useable code that can be used for Shadow Mapping**
 - Focus on how can/would OpenGL be Abstracted for real applications like a Game Engine.
 - Abstraction – need to move around code into classes for:
 - Vertex Buffer Setup
 - Index Buffer Setup
 - Vertex Array Object Setup
 - Shader Setup
 - New Renderer Setup – give it a glDraw command it will render the specified object(s)
 - Future – Texture(s), Frame Buffer(s), Material(s), and Render States (Blender on/off)
 - Today: Focus on Vertex Buffer Class and Index Buffer Class b/c we decided to use Vertex Arrays where we DO NOT have to worry about the Layout of an actual Vertex Buffer or Index Buffer, just worry about the Data.
 - Next (3) Episodes: Focus on the Vertex Arrays, Shaders and the New Renderer
- **Abstract Classes: (4:45)**
 - Create Renderer Class: (5:55)
 - Show All Files > RT+CLK > src > Add New Item ... > Select Header File (.h) > Tab > Renderer.h > Add
 - RT+CLK > src > Add New Item ... > Select C++ File (.cpp) > Tab > Renderer.cpp > Add
 - Cut & Paste code from Application.cpp to Renderer.h
 - Remove static from GLClearErrors() and GLLogCall functions to make them declarations and move their definitions to Renderer.cpp
 - New Renderer.h code: (7:55)

```
#pragma once
```

```
#include <GL/glew.h> /* must be the 1st #include BEFORE ANY other #includes */
```

```
/* Vid#10: (14:30) add ASSERT(x) macro to validate a condition and call a breakpoint if true  
using the MSVC function __debugbreak() */
```

```
#define ASSERT(x) if (!(x)) __debugbreak();
```

```
/* Vid#10: (16:20) GLCall(x) macro where (x) is the call function to Clear OpenGL Error(s)  
that calls the GLClearErrors() function */
```

```
/* Vid#10 (18:45) use macros to find out which line of code this errored function occurred.  
In GLLogCall(x) - changed to a string (#x) for printing the file name (__FILE__),  
and printing the line number (__LINE__) */
```



```

#define GLCall(x) GLClearErrors();\
    x;\
    ASSERT(GLLogCall(#x, __FILE__, __LINE__))

/* Vid13: remove static from GLClearErrors() and GLLogCall functions to make them declarations
and move their definitions to Renderer.cpp */

void GLClearErrors();
bool GLLogCall(const char* function, const char* file, int line);

```

- New Renderer.cpp code: (7:55)

```

/* Vid13: contains definitions declared in Renderer.h */

#include "Renderer.h"
#include <iostream>

void GLClearErrors()
{
    /* loop while there are errors and until GL_NO_ERROR is returned */
    while (glGetError() != GL_NO_ERROR);
}

/* Vid10: GLLogCall() static function that returns a bool and accepts parameters that allow the console
to printout the C++ source file, the line of code, and the function name that errored */

bool GLLogCall(const char* function, const char* file, int line)
{
    while (GLenum error = glGetError())
    {
        std::cout << "[OpenGL Error] (" << error << ") " << function
        << " " << file << ": " << line << std::endl;
        return false;
    }
    return true;
}

```

- Application.cpp mods (8:00)

```
#include "Renderer.h"
```

– Create VertexBuffer Class: (11:00)

- RT+CLK > src > Add New Item ... > Select Header File (.h) > Tab > VertexBuffer.h > Add
- RT+CLK > src > Add New Item ... > Select C++ File (.cpp) > Tab > VertexBuffer.cpp > Add
- New VertexBuffer.h Class code: (12:00)
 - (12:30) RT+CLK > class VertexBuffer > Quick Actions and Factorings (Ctrl+) > Create Method Implementation – DOES NOT Work in my VS2019 installation. Therefore, execute the following:
 - RT+CLK > VertexBuffer constructor > Quick Actions and Factorings (Ctrl+) > Create Definition of “VertexBuffer” constructor in VertexBuffer.cpp
 - RT+CLK > ~VertexBuffer destructor > Quick Actions and Factorings (Ctrl+) > Create Definition of “VertexBuffer” destructor in VertexBuffer.cpp
 - RT+CLK > Bind > Quick Actions and Factorings (Ctrl+) > Create Definition of “Bind” method in VertexBuffer.cpp
 - RT+CLK > Unbind > Quick Actions and Factorings (Ctrl+) > Create Definition of “Unbind” method in VertexBuffer.cpp
 - (12:50) Copy (3) lines from Application.cpp into the VertexBuffer constructor in VertexBuffer.cpp, change vao to m_RendererID, change position to data, and change sizeof(position) to size.

```
/* Vid13: contains definitions declared in Renderer.h */
#include <GL/glew.h> /* must be the 1st #include BEFORE ANY other #includes */
#include <GLFW/glfw3.h>
#include "VertexBuffer.h"
#include "Renderer.h"

VertexBuffer::VertexBuffer(const void* data, unsigned int size)
{
    GLCall(glGenBuffers(1, &m_RendererID));
    GLCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererID));
    GLCall(glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW));
}
```

- (13:15) Copy the Bind call in the constructor to the Bind() and Unbind() const methods and change m_RendererID to 0 as 2nd param in Unbind() method.

```
void VertexBuffer::Bind() const
{
    GLCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererID));
}
```

```
void VertexBuffer::Unbind() const
{
    GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
}
```

- (13:30) Add GLCall(glDeleteBuffers(1, &m_RendererID)); to the destructor

```
VertexBuffer::~VertexBuffer()
{
    GLCall(glDeleteBuffers(1, &m_RendererID));
}
```

– Create IndexBuffer Class: (13:45)

- Copy & Paste ..\OpenGL\src\VertexBuffer.h and VertexBuffer.cpp in File Explorer and Rename the (2) copies as IndexBuffer.h and IndexBuffer.cpp.
 - CLK > Refresh
 - Select both new files > RT+CLK > Include in Project
 - Refactor both new files
- Refactor IndexBuffer.h by changing all references of VertexBuffer to IndexBuffer
 - Add m_Count; // count of indices
 - Change constructor to support 32-bit indices
- New IndexBuffer.h Header code: (12:00)
 - Replace (Alt + h) IndexBuffer.h ALL references of VertexBuffer to IndexBuffer
 - Modify Constructor to support 32-bit indices elements (unsigned int* size in bytes) only of (unsigned int element count)
 - Application.cpp defines (6) indices so the count = 6 and size = 24 bytes
 - Mark Bind() and Unbind() methods as const
 - Add /* Add inline GETTER GetCount() const that returns an unsigned int */

```
inline unsigned int GetCount() const {
    return m_Count;
};
```

- New IndexBuffer.cpp Implementation code: (16:00)
 - Replace (Alt + h) IndexBuffer.cpp ALL references of VertexBuffer to IndexBuffer
 - Replace old VertexBuffer constructor params with new IndexBuffer params (const unsigned int* data, unsigned int count)
 - Add ASSERT(sizeof(unsigned int) == sizeof(GLuint)); at top of constructor code
 - Replace glBufferData method's size with sizeof(count) OR count * sizeof(unsigned int)

- Replace glBindBuffer and glBufferData method's 1st param from GL_ARRAY_BUFFER to GL_ELEMENT_ARRAY_BUFFER
- Replace Bind() and Unbind() as const glBindBuffer method's 1st param from GL_ARRAY_BUFFER to GL_ELEMENT_ARRAY_BUFFER
- Add constructor initializer list with ": m_Count(count)"
- **Modify Application code: (18:30)**
 - Add (2) #include "VertexBuffer.h", "IndexBuffer.h"
 - **Modify Application.cpp Implementation code: (18:30) to support new VertexBuffer Class**
 - /* Vid#13: (18:40) Delete OR Comment Out the original VertexBuffer creation code, move to the new VertexBuffer Class, and replace with: */

```
//VertexBuffer vb(positions, 4 * 2 * sizeof(float)); // note this code produced only the bottom triangle
VertexBuffer vb(positions, sizeof(positions)); // Replaced (22:00) and now shows both triangles

//unsigned int buffer;
//GLCall(glGenBuffers(1, &buffer));
//GLCall(glBindBuffer(GL_ARRAY_BUFFER, buffer));
//GLCall(glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW));
```
 - Theoretically, this deleted/uncommented code Binds the VertexBuffer automatically in the new VertexBuffer Class constructor so we don't need to and NEVER Unbind it. Therefore, we don't need to call vb.Bind() because it was automatically bound in the constructor.
 - If we create multiple VertexBuffer(s), we'll have to Re-Bind the ones we use BUT it will be handled by the creation of the Vertex Array Abstraction.
 - **Modify Application.cpp Implementation code: (19:30) to support new IndexBuffer Class**
 - /* Vid#13: (19:35) Delete OR Comment Out IndexBuffer creation code, move to the new IndexBuffer Class, and replace with: */

```
/* Vid#13: (22:45) Stack allocated object, its destructor is called when the scope exits
which is called at the end of the main() function */

IndexBuffer ib(indices, 6);

//unsigned int ibo;
//GLCall(glGenBuffers(1, &ibo));
//GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo));
//GLCall(glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW));
```
- **Modify Rendering Loop code: (20:00)**
 - /* Vid#13: (20:00) delete OR comment out this glBindBuffer and replace with: */

```
ib.Bind(); // new bind call to Index Buffer Class method
```

```
//GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo)); // old bind method Index Buffer (ibo)
```

- **Run & Test (F5) (22:00) – Drew Only the Bottom Triangle. The Top Triangle was NOT Drawn but showed the black background.**

- **Root Cause:** `//VertexBuffer vb(positions, 4 * 2 * sizeof(float));` // note this code produced only the bottom triangle
- **Solution:** `VertexBuffer vb(positions, sizeof(positions));` // Passes! Now Draws both triangles

- **Console Window DOES NOT Close AFTER Closing Application Window (Rectangle): (22:10)**

- **CLK > Debugger Pause (| |) Break All (Ctrl + Alt + Break): View CallStack**

- **Renderer.cpp OpenGL.exe!GLClearErrors() Line 9 C++**

```
void GLClearErrors()
{
    /* loop while there are errors and until GL_NO_ERROR is returned */
    while (glGetError() != GL_NO_ERROR);
}
```

- **Problem:** `GLClearErrors()` is stuck in an infinite loop coming from `IndexBuffer` destructor which is coming from `main()` because it's trying to cleanup all of the stack allocated objects (e.g. `IndexBuffer`).
- **Root Cause:** **Application.cpp**
 - `IndexBuffer ib(indices, 6);` /* Vid#13: (22:45) Stack allocated object, its ~destructor is called when the scope exits which is called at the end of the `main()` function */
 - `glfwTerminate();` /* OpenGL GLFW Terminate destroys OpenGL context BEFORE `IndexBuffer` ~destructor is Called. We now DO NOT have an OpenGL Context */
 - `while (glGetError() != GL_NO_ERROR);` /* loop while there are errors and until `GL_NO_ERROR` is returned */
 - **Call `glGetError()` will return a `glGetError()` IF THERE IS NO CONTEXT! Therefore, we are STUCK in an Infinite Loop of an OpenGL Comedy!**
- **Solution(s):**
 - **Heap Allocate the Vertex Array (VertexBuffer AND IndexBuffer) BEFORE `glfwTerminate()`;**
 - **In `Application.cpp`, Create a New Scope {...} from**
- **Solution Code Modifications: (24:15)**

```
/* Create a New Scope {...} FROM Here TO BEFORE glfwTerminate(); */
{
    float positions[] = {
        -0.5f, -0.5f, 0.0f, // vertex 0
        0.5f, -0.5f, 0.0f, // vertex 1
        0.5f, 0.5f, 0.0f, // vertex 2
        -0.5f, 0.5f, 0.0f, // vertex 3
    };
};
```

```

        /* delete Shader */
        GLCall(glDeleteProgram(shader));
    }

    /* OpenGL GLFW Terminate destroys OpenGL context BEFORE IndexBuffer ~destructor is Called
       We now DO NOT have an OpenGL Context */
    glfwTerminate();
    return 0;
}

```

- **Tools > Options > Debugging > General > Checkbox > Automatically close the console when debugging stops.**
- **Run & Test (F5): (24:40) – Closing Application Window NOW Closes Console Window**

– **Future Application Example: Space Shuttle (20:15)**

- Create one(1) Vertex Buffer that includes ALL Vertices in the entire model
- Create multiple Index Buffers drawing parts of the ship (wings, cockpit glass, tires, etc.), with each Index Buffer dedicated to a section specifying (material, texture, color, etc.). Index Buffers contain starting offsets, indices counts, etc) as pointers into the Vertex Buffer for each unique ship part to draw.
 - Option of Tie Draw Call(s) to Index Buffer(s) OR
 - **Let the new Renderer to handle the Draw Call(s). (21:30)**
- Expand the new Renderer Class (21:35) will be supplied with an IndexBuffer (asking for count), VertexBuffer, and issue the glDrawElements Call itself.

▪ **The End**

Vid#14: Buffer Layout Abstraction in OpenGL

▪ Concepts:

- Why should we use a Vertex Array and Abstract into its own Class?
 - This is the list of features I need, these are my requirements, and then build a Class to facilitate them.
- What is the goal and purpose of a Vertex Array Abstraction?
 - Tie together a VertexBuffer (just a data buffer of bytes with NO concept of data types, sizes, etc.) with an actual Layout (that defines types, sizes, functions of groups of bytes within the VertexBuffer).
 - Option of using IndexBuffers
- Vertex Array API Creation Steps: (2:40)
 - Create Vertex Array (series of buffers) Object(s) – VAO(s)
 - Tell the Vertex Array to use this Layout where its definition is created on the CPU side but executes on the GPU side.
 - Example: Picking Cache – read each triangle of our model so when a user clicks somewhere on the screen, it selects that location's object and fires an Array to check if it intersects a triangle in the model.
 - You might want to store the VertexBuffer on the CPU side to perform the triangle collision test.
 - Add a VertexBuffer to it
 - Call VertexArray.Bind() on the VAO(s)
- Buffer Layout Abstraction code changes (5:00)
 - Modify Application.cpp code:
 - Create new VertexArray and BufferLayout code AFTER Creating the Vertex Array Object (VAO)
- Create New VertexArray Class (7:30)
 - RT+CLK > src > Add > Class > Class Name: "VertexArray" >
 - New VertexArray.h Class code: (7:40)
 - Create Constructor, ~Destructor, void AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout);
 - The index of each element as it appears in the actual vector. (16:30)
 - When it comes time to bind a layout with a vertex buffer, we set it up in the new VertexArray class.
 - Modify VertexArray.h as follows: (17:00)
 - Create Implementations for VertexArray constructor, ~destructor, and AddBuffer method
 - `/* Vid#14: (17:30) comment out GLCall(glEnableVertexAttribArray(0));, GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0)); in Application.cpp and move to VertexArray.cpp */`

```
#pragma once

#include "VertexBuffer.h"
#include "VertexBufferLayout.h"

class VertexArray
{
```

```

private:
    unsigned int m_RendererID;

public:
    /* VertexArray constructor */
    VertexArray();

    /* VertexArray ~destructor */
    ~VertexArray();

    /* AddBuffer takes in 2 const params:
        1st - VertexBuffer& vb,
        2nd - VertexBufferLayout &, layout */
    void AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout);

    void Bind() const;
    void Unbind() const;
};

```

- **New VertexArray.cpp implementation code: (17:10)**

- **New VertexArray.cpp Implementation Code:**

```

#include "VertexArray.h"

#include "Renderer.h"

VertexArray::VertexArray()
{
    GLCall(glGenVertexArrays(1, &m_RendererID));
}

VertexArray::~VertexArray()
{
    GLCall(glDeleteVertexArrays(1, &m_RendererID));
}

/* Bind the Vertex Buffer and Setup the Vertex Layout */
void VertexArray::AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout)
{
    /* Bind the VertexArray */
    Bind();

    /* Bind the VertexBuffer */
    vb.Bind();

    /* Setup a Vertex Layout */
    const auto& elements = layout.GetElements();
    unsigned int offset = 0;
}

```



```

for (unsigned int i = 0; i < elements.size(); i++)
{
    const auto& element = elements[i];

    /* Enable or disable a generic vertex attribute array for index = 0 */
    GLCall(glEnableVertexAttribArray(i));

    /* When we specify the GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0));
    ,
        index 0 (1st param) of this Vertex Array is going to be bound to the currently bound
        glBindBuffer(GL_ARRAY_BUFFER, buffer).

    define an array of generic vertex attribute data
        1st param index = i,
        2nd param size = element.count for a (3) component vector that represents each Vertex position,
        3rd param symbolic constant = element.type,
        4th param normalized = converted directly as fixed-point values (GL_FALSE) 4th param,
        5th param stride = the amount of bytes between each Vertex based on
            2nd param vec2 (x, y, z) component position vector of 3 floats = 12 bytes,
        6th param pointer = position has an offset pointer = 0 */

    GLCall(glVertexAttribPointer(i, element.count, element.type,
        element.normalized, layout.GetStride(), (const void*)offset));

    offset += element.count * VertexBufferElement::GetSizeOfType(element.type);
}
}

void VertexArray::Bind() const
{
    GLCall(glBindVertexArray(m_RendererID));
}

void VertexArray::Unbind() const
{
    GLCall(glBindVertexArray(0));
}

```

– Create New VertexBufferLayout Class (10:00)

- RT+CLK > src > Add > Class > Class Name: “VertexBufferLayout” > **Ok**
- New VertexBufferLayout.h Class code: (10:10)
 - Create Constructor, ~Destructor, struct VertexBufferElement, and class VertexBufferLayout

```

#pragma once
#include <vector>
#include "Renderer.h"

struct VertexBufferElement
{
    unsigned int type;
    unsigned int count;
    unsigned char normalized;

    /* return size of type */
    static unsigned int GetSizeOfType(unsigned int type)
    {
        switch (type)
        {
            case GL_FLOAT:                return 4;
            case GL_UNSIGNED_INT:         return 4;
            case GL_UNSIGNED_BYTE:        return 1;
        }
        ASSERT(false);
        return 0;
    }
};

class VertexBufferLayout
{
private:
    /* VertexBufferElement vector - m_Elements */
    std::vector<VertexBufferElement> m_Elements;
    unsigned int m_Stride;

public:
    /* VertexBufferLayout constructor */
    VertexBufferLayout()
        : m_Stride(0) {}

    /* VertexBufferLayout ~destructor */

    /* create template<typename T> */
    template<typename T>

    void Push(unsigned int count)
    {
        static_assert(false);
    }
}

```

```

/* create template<> for float */
template<>
void Push<float>(unsigned int count)
{
    m_Elements.push_back({ GL_FLOAT, count, GL_FALSE });
    m_Stride += count * VertexBufferElement::GetSizeOfType(GL_FLOAT);
}

/* create template<> for unsigned int */
template<>
void Push<unsigned int>(unsigned int count)
{
    m_Elements.push_back({ GL_UNSIGNED_INT, count, GL_FALSE });
    m_Stride += count * VertexBufferElement::GetSizeOfType(GL_UNSIGNED_INT);
}

/* create template<> for unsigned char */
template<>
void Push<unsigned char>(unsigned int count)
{
    m_Elements.push_back({ GL_UNSIGNED_BYTE, count, GL_TRUE });
    m_Stride += count * VertexBufferElement::GetSizeOfType(GL_UNSIGNED_BYTE);
}

/* add inline method to get elements that returns const& */
inline const std::vector<VertexBufferElement> GetElements() const { return m_Elements; }

/* add inline method to get stride */
inline unsigned int GetStride() const { return m_Stride; }
};

```

- **New VertexBufferLayout.cpp Implementation Code: (10:10)**

- **Includes: VertexBufferLayout.h**

```
#include "VertexBufferLayout.h"
```

- **Modifications to Application.cpp: (23:50)**

- **Include VertexArray.h, Make VertexBufferLayout, layout.Push<float<(3), va.AddBuffer(vb, layout).**

- **Updated Application.cpp Code: (**

```

#include <GL/glew.h> /* must be the 1st #include BEFORE ANY other #includes */
#include <GLFW/glfw3.h>
#include <iostream>
/* Vid#8 add includes to read, parse C++ external file: Basic.shader, and add to each Shader buffer */
#include <fstream>
#include <string>
#include <sstream>

```

```

/* Vid#13: (8:00) add include "Renderer.h" */
#include "Renderer.h"

/* Vid#13: (18:30) Add #includes for two new classes */
#include "VertexBuffer.h"
#include "IndexBuffer.h"

/* Vid#14: (23:50) Add #includes for new class */
#include "VertexArray.h"

/** Create a struct that allows returning multiple items */
struct ShaderProgramSource
{
    std::string VertexSource;
    std::string FragmentSource;
};

/** Vid#8 Add new function ParseShader to parse external Basic.shader file
    returns - struct ShaderProgramSource above which contains two strings (variables)
    note: C++ functions are normally capable of only returning one variable */
static ShaderProgramSource ParseShader(const std::string& filepath)
{
    /* create enum class for each Shader type */
    enum class ShaderType
    {
        NONE = -1, VERTEX = 0, FRAGMENT = 1
    };

    /* open file */
    std::ifstream stream(filepath);

    /* define buffers for 2 Shaders: vertexShader and fragmentShader */
    std::stringstream ss[2];

    /* set initial ShaderType = NONE */
    ShaderType type = ShaderType::NONE;

    /* parse file line by line */

```

```

std::string line;

while (getline(stream, line))
{
    /* find "#shader" keyword */
    if (line.find("#shader") != std::string::npos)
    {
        if (line.find("vertex") != std::string::npos)
            /* set mode to vertex */
            type = ShaderType::VERTEX;

        else if (line.find("fragment") != std::string::npos)
            /* set mode to fragment */
            type = ShaderType::FRAGMENT;
    }
    else if (type != ShaderType::NONE)
        /* add each line to the corresponding buffer after detecting the ShaderType */
        {
            /* type is an index to push data into the selected array buffer, casted to a Shader int type,
            to add each new line plus newline char */

            ss[(int)type] << line << '\n';
        }
    else
    {
        /* Got non-introductory line out of sequence! Don't know what type to use! Consider asserting,
        or throwing an exception, or something, depending on how defensive you
        want to be with respect to the input file format. */
    }
}

/* returns a struct comprised of two ss strings */
return { ss[0].str(), ss[1].str() };
}

/** Vid#7: create static int CreateShader function with parameters:
    unsigned int type (used raw C++ type instead of OpenGL GLuint type to allow other non-OpenGL GPU driver
    implementations),

    const std::string& source
    returns a static unsigned int, takes in a type and a string ptr reference to a source */

static unsigned int CompileShader(unsigned int type, const std::string& source)
{
    /* change GL_VERTEX_SHADER to type */

    unsigned int id = glCreateShader(type);

```

```

/* returns a char ptr* src to a raw string (the beginning of our data)
   assigned to source which needs to exist before this code is executed
   pointer to beginning of our data */

const char* src = source.c_str();

/* specify glShaderSource(Shader ID, source code count, ptr* to memory address of ptr*, length)
   as the source of our Shader */

glShaderSource(id, 1, &src, nullptr);

/* specify glCompileShader(Shader ID), then return the Shader ID */

glCompileShader(id);

/*error handling - query void glGetShaderiv(GLuint shader, GLenum pname, GLint *params);
   i - specifies an integer
   v - specifies a vector (array) */

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

if (result == GL_FALSE)
{
    /* query message - length and contents
       void glGetShaderiv(GLuint shader, GLenum pname, GLint *params); */

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

    /* construct char message[length] array allocated on the stack */
    char* message = (char*)alloca(length * sizeof(char));

    /* glGetShaderInfoLog - Returns the information log for a shader object
       void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infoLog); */

    glGetShaderInfoLog(id, length, &length, message);

    /* print the message to the console using std::cout */

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

    /* delete Shader using id and return error code = 0 */

    glDeleteShader(id);

```

```

        return 0;
    }
    return id;
}

/** Vid#7: create static int CreateShader function with parameters:
    const string pointer vertexShader(actual source code),
    const string pointer fragmentShader (actual source code)
    returns a static int, takes in the actual source code of these two Shader strings ***/

static unsigned int CreateShader(const std::string& vertexShader, const std::string& fragmentShader)
{
    /* glCreateProgram() return an unsigned int program */

    unsigned int program = glCreateProgram();

    /* create vertexShader object */

    unsigned int vs = CompileShader(GL_VERTEX_SHADER, vertexShader);

    /* create fragmentShader object */

    unsigned int fs = CompileShader(GL_FRAGMENT_SHADER, fragmentShader);

    /* attach vs and fs Shader files, link and validate them to our program ID
        void glAttachShader(GLuint program, GLuint shader); */

    glAttachShader(program, vs);
    glAttachShader(program, fs);

    /* void glLinkProgram(GLuint program); */

    glLinkProgram(program);

    /* void glValidateProgram(    GLuint program); */

    glValidateProgram(program);

    /* finally, delete the intermediary *.obj files (objects vs and fs) of program ID
        and return an unsigned int program
        void glDeleteShader(GLuint shader); */

    glDeleteShader(vs);
    glDeleteShader(fs);

    return program;
}

```

```

int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */

    if (!glfwInit())
        return -1;

    /* Vid#12 (8:10) OpenGL GLFW Version 3.3 create an Open Context and Window with the Core Profile
       GLFW_OPENGL_CORE_PROFILE
       Note: ONLY (GLFW_CONTEXT_VERSION_MAJOR, 2) and (GLFW_CONTEXT_VERSION_MINOR, 1) WORKS!!!
       All other combinations of ints (e.g. 2, 3) of later major/minor versions Fails
       with the following console output msg:

       C:\Dev\Cherno\OpenGL\bin\Win32\Debug\OpenGL.exe (process 4936) exited with code -1.
       To automatically close the console when debugging stops,
       enable Tools->Options->Debugging->Automatically close the console when debugging stops.
       Press any key to close this window . . .
    */

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
    glfwWindowHint(GLFW_OPENGL_ANY_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    /***** Create a windowed mode window and its OpenGL context
        glfwCreateWindow MUST BE PERFORMED BEFORE ANY glfwWindowHint(s) *****/

    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

    if (window==NULL)
    {
        return -1;
    }

    /*** Make the window's context current - this MUST BE PERFORMED BEFORE glewInit() !!! ***/

    glfwMakeContextCurrent(window);

    /* Vid#11 (9:00) - should sync our Swap with the monitor's refresh rate
       and produce a smooth color change transition */

    GLCall(glfwSwapInterval(1));

    /*** Vid#3: JT Added Modern OpenGL code here - MUST FOLLOW glfwMakeContextCurrent(window) ***/

```



```

if (glewInit() != GLEW_OK)
{
    std::cout << "glewInit() Error!" << std::endl;
}

/** Vid#3: JT Added Print Modern OpenGL Version code here **/

std::cout << glGetString(GL_VERSION) << std::endl;

/* Vid#9B: Vertex Buffer - remove 2 duplicate vertices of the 6 vertices in position[] to implement
an Index Buffer */
/* Create a New Scope {...} FROM Here TO BEFORE glfwTerminate(); */
{
    float positions[] = {
        -0.5f, -0.5f, 0.0f, // vertex 0
        0.5f, -0.5f, 0.0f, // vertex 1
        0.5f, 0.5f, 0.0f, // vertex 2
        -0.5f, 0.5f, 0.0f, // vertex 3
    };

    /* Vid9B: create Index Buffer using new indices[] array
note: must be unsigned but can use char, short, int, etc. */

    unsigned int indices[] = {
        0, 1, 2, // 1st right triangle drawn CCW
        2, 3, 0 // 2nd inverted right triangle drawn CCW
    };

    /* Vid#12: (10:00) Create Vertex Array Object (vao) BEFORE Creating Vertex Buffer (buffer)
Create unsigned int Vertex Array Object ID: vao
GLCall(glGenVertexArrays(numVAs, stored vao IDrefptr))
GLCall(glBindVertexArray(VAO ID)) */

/* Vid#14: (28:50) author Deletes this create VertexArray object code - Run (F5) WORKS */
/* create VertexArray object vao */
//unsigned int vao;
/* copy & paste in VertexArray.cpp and comment out */
//GLCall(glGenVertexArrays(1, &vao));
//GLCall(glBindVertexArray(vao));

/* Vid#14: (5:00) create VertexArray va and VertexBuffer vb AFTER creating Vertex Array Object (vao)
*/

VertexArray va;
//VertexBuffer vb(positions, 4 * 2 * sizeof(float));
VertexBuffer vb(positions, sizeof(positions));

```

```

/* Vid#14: (24:00) create VertexBufferLayout */

VertexBufferLayout layout;
layout.Push<float>(3);
va.AddBuffer(vb, layout);

/* Vid#13: (19:35) Delete OR Comment Out IndexBuffer creation code,
   move to the new IndexBuffer Class, and replace with: */

IndexBuffer ib(indices, 6);

ShaderProgramSource source = ParseShader("res/shaders/Basic.shader");

std::cout << "VERTEX" << std::endl;
std::cout << source.VertexSource << std::endl;
std::cout << "FRAGMENT" << std::endl;
std::cout << source.FragmentSource << std::endl;

/* Call to create vertexShader and fragmentShader above */

unsigned int shader = CreateShader(source.VertexSource, source.FragmentSource);

/* Vid#11 (3:40) Bind our Shader - now wrapped in GLCall() to check if shader was found */

GLCall(glUseProgram(shader));

/* Retrieve the int location of the variable location */

GLCall(int location = glGetUniformLocation(shader, "u_Color"));

/* Check if location DID NOT return -1 */

GLCall(ASSERT(location != -1));

/* Determine the type of Data to send to the GPU's Shader - (4) floats
   1st param - int location of these (4) floats */

GLCall(glUniform4f(location, 0.8f, 0.3f, 0.8f, 1.0f));

/* Vid#14: (29:00) author Deletes GLCall(glBindVertexArray(0));
   and Add va.Unbind() - Run (F5) WORKS */
//GLCall(glBindVertexArray(0));

va.Unbind();

/* Vid#12: (4:00) Unbind the Shader (shader), the vertex buffer (buffer), and the index buffer (ibo)
   by setting each = 0 and re-bind all (3) inside the Rendering while loop before the glDraw cmd */

```

```

/* Vid#12 (11:10) add clear glBindVertexArray(0) binding */

GLCall(glUseProgram(0));
GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0));

/* Vid#11 (8:00) - Animate Loop: 1st define 4 float variables: r, g, b, and i */
float r = 0.0f;           // red color float var initially set to zero
float increment = 0.05f;   // color animation float increment var initially set to 0.05

/* Games Render Loop until the user closes the window */

while (!glfwWindowShouldClose(window))
{
    /* Render here */

    GLCall(glClear(GL_COLOR_BUFFER_BIT));

    /* Vid#12: (4:45) Bind Shader (shader), Uniform (location), Vertex Buffer (buffer)
       and Index Buffer (ibo) BEFORE calling glDrawElements... */

    GLCall(glUseProgram(shader));           // bind our shader
    GLCall(glUniform4f(location, r, 0.3f, 0.8f, 1.0f)); // setup uniforms

    /* Vid#12: (11:15) new method - add binding GLCall(glBindVertexArray(vao));
       and then bind Index Array Buffer (ibo) */

    /* Vid#14: (24:00) Delete OR Comment out GLCall(glBindVertexArray(vao)); */
    //GLCall(glBindVertexArray(vao)); // bind our Vertex Array Object
    /* and Replace with va.Bind() Call */

    va.Bind();

    /* Vid#13: (20:00) delete OR comment out this glBindBuffer and replace with: */

    ib.Bind(); // new bind call to Index Buffer Class method
    GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr)); // Draw Elements call

    /* Vid#11 (8:30) check if r value > 1.0f --> set increment = -0.05f
       else if r value < 0.0f --> set increment = 0.05f */

    if (r > 1.0f)
        increment = -0.05f;
    else if (r < 0.0f)
        increment = 0.05f;
    r += increment;
}

```

```

        /* Swap front and back buffers */
        GLCall(GLFW_SWAPBUFFERS(window));

        /* Poll for and process events */
        GLCall(GLFW_POLL_EVENTS());
    }

    /* delete Shader */
    GLCall(GL_DELETE_PROGRAM(shader));
}

/* OpenGL GLFW Terminate destroys OpenGL context BEFORE IndexBuffer ~destructor is Called
   We now DO NOT have an OpenGL Context */

glfwTerminate();
return 0;
}

```

- **Run & Test (F5) (26:35) – Failed! Instead of the Rectangle, I get a funky shape.**
 - Problem: Vid#14 (24:00), author entered `layout.Push<float>(2);` instead of (3)
 - Solution: change to (3), Run & Test (F5) – Passed! Draws correct Rectangle
- Clean up Application Code:

Vid#15: Shader Abstraction in OpenGL

- **Concepts:**
 - **How can we Abstract Shaders?**
 - Shaders are an incredibly complex topic but are very important to graphics programming or any kind of Rendering.
 - Shader generation and creation on the fly during runtime is common in game engines.
 - In OpenGL, it's much simpler than developing a game engine. You just write text in a shader file or as a string and bind it.
 - **Goal for this lesson:**
 - Take the OpenGL Shader code previously written, abstract it out behind a new easy to use API that keeps the client side code concise and keeps all of the glShader code details separate.
 - **What specifically are the Steps required to Abstract, Create, Compile, and Apply Shader code?**
 - Pass in a string of shader code inside a file that is to be compiled into a Shader
 - Bind and Unbind the Shaders for Use
 - Set all of the different Uniforms for the Shader
 - Future: reading back attributes (numUniforms, etc.) back from the Shader and passing shader source code to the Shader (game engines)
 -
- **Refactor Source code for implementing Abstract Shaders (4:40)**
 - **Add New Class: Shader**
 - **New Header Shader.h code: (5:30)**
 - Create Class Shader with constructor taking 2 params: (const string&, filepath), ~destructor, Bind(), Unbind(), and SetUniform4f taking 5 params: (const std::string& name, float v0, float v1, float v2, float v3)
 - Add private attributes: m_FilePath and m_RendererID
 - Add private method declaration: GetUniformLocation used to retrieve OpenGL Uniform Locations and Refactor definition in Shader.cpp
 - Add private method declaration: `static unsigned int CreateShader(const std::string& vertexShader, const std::string& fragmentShader);`
 - Add private method declaration: `static unsigned int CompileShader(unsigned int type, const std::string& source);`
 - Add private method declaration: `static ShaderProgramSource ParseShader(const std::string& filepath);`
 - Move `struct ShaderProgramSource` that allows returning multiple items from Application.cpp to Shader.h
 - Delete static from all methods

```
#pragma once

#include <string>

/* Vid#15: (17:40) include hash table (unordered_map) */
#include <unordered_map>
```

```

/** Vid#15: (9:55) Move struct ShaderProgramSource that allows returning multiple items
    from Application.cpp to Shader.h */

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

class Shader
{
private:
    /* save file path */
    std::string m_FilePath;

    /* save attributes */
    unsigned int m_RendererID;

    /* Vid#15:(17:40) caching for UniformLocationCache */
    std::unordered_map<std::string, int> m_UniformLocationCache;

public:
    /* Shadow constructor */
    Shader(const std::string& filepath);

    /* Shadow destructor */
    ~Shader();

    /* Shadow Bind method to execute glUseProgram */
    void Bind() const;

    /* Shadow Unbind method */
    void Unbind() const;

    /* Set Uniform 4f with string refptr to name and 4 floats */
    void SetUniform4f(const std::string& name, float v0, float v1, float v2, float v3);

private:
    /** Vid#15:(9:45) Move ParseShader method declaration to Shader.cpp to parse ext. Basic.shader file
        returns - struct ShaderProgramSource above which contains two strings (variables)
        note: C++ functions are normally capable of only returning one variable */

    ShaderProgramSource ParseShader(const std::string& filepath);

    /** Vid#15: (9:45) Move CreateShader method declaration with parameters:
        const string pointer vertexShader(actual source code),
        const string pointer fragmentShader (actual source code)
        returns a static int, takes in the actual source code of these two Shader strings */

```

```

        unsigned int CreateShader(const std::string& vertexShader, const std::string& fragmentShader);

        /*** Vid#15: (9:45) Move CompileShader function with parameters:
            unsigned int type (used raw C++ type instead of OpenGL GLuint type to allow other non-OpenGL
            GPU driver implementations),
            const std::string& source
            returns a static unsigned int, takes in a type and a string ptr reference to a source ***/

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

        /* Vid#15: (9:30) Declare GetUniformLocation used to retrieve OpenGL Uniform Locations
            and Refactor definition in Shader.cpp */

        unsigned int GetUniformLocation(const std::string& name);
    };

```

- **New Implementation Shader.cpp code: (9:20)**

- Includes: Shader.h and Refactor all method declarations in Shader.h to definitions in Shader.cpp
- Code Refactored GetUniformLocation method from Shader.h declaration
- Move CreateShader() Function from Application.cpp into the new Shader.cpp
- Move CompileShader() Function from Application.cpp into the new Shader.cpp
- Move ParseShader() from Application.cpp into the new Shader.cpp
- Delete static from and add Shader:: member to (3) methods

```

#include "Shader.h"

/*** Vid#15: (10:00) Added required include libraries ***/

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "Renderer.h"

/* Shader constructor */
Shader::Shader(const std::string& filepath)
    /* m_FilePath(filepath) is used for debugging purposes */
    : m_FilePath(filepath), m_RendererID(0)
{
    /* Vid#15: (12:15) Moved to Shader.cpp constructor */
    ShaderProgramSource source = ParseShader(filepath);

    /* Vid#15 (12:30) Call to create vertexShader and fragmentShader above now using m_RendererID */
    m_RendererID = CreateShader(source.VertexSource, source.FragmentSource);
}

```

```

/* Shader destructor */
Shader::~Shader()
{
    /* Vid#15 (12:30) destroys */
    GLCall(glDeleteProgram(m_RendererID));
}

void Shader::Bind() const
{
    /* Vid#15 (13:00) Shader Bind method */
    GLCall(glUseProgram(m_RendererID));
}

void Shader::Unbind() const
{
    /* Vid#15 (13:00) Shader Unind method */
    GLCall(glUseProgram(0));
}

void Shader::SetUniform4f(const std::string& name, float v0, float v1, float v2, float v3)
{
    /* Vid#15: (13:15) pass in location of the Uniform */
    GLCall(glUniform4f(GetUniformLocation(name), v0, v1, v2, v3));
}

unsigned int Shader::GetUniformLocation(const std::string& name)
{
    /* Vid#15: (18:00) Check if UniformLocationCache contains the name
       and if it does, return the location, else GetUniformLocation() */

    if (m_UniformLocationCache.find(name) != m_UniformLocationCache.end())
        return m_UniformLocationCache[name];

    /* Vid#15: (13:45) should be int location like in Application.cpp */

    GLCall(int location = glGetUniformLocation(m_RendererID, name.c_str()));

    /* test for valid (-1) for Shader location when a Uniform is declared but not used yet */

    if (location == -1)
        std::cout << "Warning: uniform '" << name << "' doesn't exist!" << std::endl;

    /* cache the location improves the performance with multiple uniforms */
    m_UniformLocationCache[name] = location;

    /* return location */
    return location;
}

```



```

/** Vid#7: create static int CompileShader function with parameters:
    unsigned int type (used raw C++ type instead of OpenGL GLuint type to allow other non-OpenGL GPU driver
    implementations),
    const std::string& source
    returns a static unsigned int, takes in a type and a string ptr reference to a source */

unsigned int Shader::CompileShader(unsigned int type, const std::string& source)
{
    /* change GL_VERTEX_SHADER to type */
    unsigned int id = glCreateShader(type);

    /* returns a char ptr* src to a raw string (the beginning of our data)
       assigned to source which needs to exist before this code is executed
       pointer to beginning of our data */

    const char* src = source.c_str();

    /* specify glShaderSource(Shader ID, source code count, ptr* to memory address of ptr*, length)
       as the source of our Shader */

    glShaderSource(id, 1, &src, nullptr);

    /* specify glCompileShader(Shader ID), then return the Shader ID */

    glCompileShader(id);

    /*error handling - query void glGetShaderiv(GLuint shader, GLenum pname, GLint *params);
       i - specifies an integer
       v - specifies a vector (array) */
    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);

    if (result == GL_FALSE)
    {
        /* query message - length and contents
           void glGetShaderiv(GLuint shader, GLenum pname, GLint *params); */

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

        /* construct char message[length] array allocated on the stack */

        char* message = (char*)alloca(length * sizeof(char));

        /* glGetShaderInfoLog - Returns the information log for a shader object
           void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infoLog); */
        glGetShaderInfoLog(id, length, &length, message);
    }
}

```

```

        /* print the message to the console using std::cout */

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

        /* delete Shader using id and return error code = 0 */

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

/** Vid#7: create static int CreateShader function with parameters:
    const string pointer vertexShader(actual source code),
    const string pointer fragmentShader (actual source code)
    returns a static int, takes in the actual source code of these two Shader strings **/
/* Vid#15: (9:20) Moved CreateShader from Application.cpp to Shader.cpp, then Refactored */

unsigned int Shader::CreateShader(const std::string& vertexShader, const std::string& fragmentShader)
{
    /* glCreateProgram() return an unsigned int program */

    GLuint program = glCreateProgram();

    /* create vertexShader object */

    unsigned int vs = CompileShader(GL_VERTEX_SHADER, vertexShader);

    /* create fragmentShader object */

    unsigned int fs = CompileShader(GL_FRAGMENT_SHADER, fragmentShader);

    /* attach vs and fs Shader files, link and validate them to our program ID
        void glAttachShader(GLuint program, GLuint shader); */

    glAttachShader(program, vs);
    glAttachShader(program, fs);

    /* void glLinkProgram(GLuint program); */
    glLinkProgram(program);

    /* void glValidateProgram(    GLuint program); */
    glValidateProgram(program);
}

```

```

/* finally, delete the intermediary *.obj files (objects vs and fs) of program ID
   and return an unsigned int program
   void glDeleteShader(GLuint shader); */

glDeleteShader(vs);
glDeleteShader(fs);

return program;
}

/**/ Vid#8 Add new function ParseShader to parse external Basic.shader file
returns - struct ShaderProgramSource above which contains two strings (variables)
note: C++ functions are normally capable of only returning one variable ***/

ShaderProgramSource Shader::ParseShader(const std::string& filepath)
{
    /**/ Vid#15: (10:20) added - should have been there in Vid#14 ***/
    std::ifstream stream(filepath);

    /* create enum class for each Shader type */
    enum class ShaderType
    {
        NONE = -1, VERTEX = 0, FRAGMENT = 1
    };

    /* define buffers for 2 Shaders: vertexShader and fragmentShader */
    std::stringstream ss[2];

    /* set initial ShaderType = NONE */
    ShaderType type = ShaderType::NONE;

    /* parse file line by line */
    std::string line;
    while (getline(stream, line))
    {
        /* find "#shader" keyword */
        if (line.find("#shader") != std::string::npos)
        {
            if (line.find("vertex") != std::string::npos)
            {
                /* set mode to vertex */
                type = ShaderType::VERTEX;
            }
            else if (line.find("fragment") != std::string::npos)
            {
                /* set mode to fragment */
                type = ShaderType::FRAGMENT;
            }
        }
        else if (type != ShaderType::NONE)
            /* add each line to the corresponding buffer after detecting the ShaderType */

```

```

    {
        /* type is an index to push data into the selected array buffer, casted to a Shader int type,
           to add each new line plus newline char */

        ss[(int)type] << line << '\n';
    }
    else
    {
        /* Got non-introductory line out of sequence! Don't know what type to use! Consider asserting,
           or throwing an exception, or something, depending on how defensive you
           want to be with respect to the input file format. */
    }
}

/* returns a struct comprised of two ss strings */

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

```

➤ **Modification to Application.cpp code:**

○ **Moved CreateShader(), CompileShader(), ParseShader() methods to Shader.cpp**

```

#include <GL/glew.h> /* must be the 1st #include BEFORE ANY other #includes */
#include <GLFW/glfw3.h>
#include <iostream>

/* Vid#8 add includes to read, parse C++ external file: Basic.shader, and add to each Shader buffer */
#include <fstream>
#include <string>
#include <sstream>

/* Vid#13: (8:00) add include "Renderer.h" */
#include "Renderer.h"

/* Vid#13: (18:30) Add #includes for two new classes */
#include "VertexBuffer.h"
#include "IndexBuffer.h"

/* Vid#14: (23:50) Add #includes for new VertexArray class */
#include "VertexArray.h"

/* Vid#15: (15:40) Add #includes for new Shader class */
#include "Shader.h"

int main(void)
{
    GLFWwindow* window;

```

```

/* Initialize the library */
if (!glfwInit())
    return -1;

/* Vid#12 (8:10) OpenGL GLFW Version 3.3 create an Open Context and Window with the Core Profile
   GLFW_OPENGL_CORE_PROFILE
   Note: ONLY (GLFW_CONTEXT_VERSION_MAJOR, 2) and (GLFW_CONTEXT_VERSION_MINOR, 1) WORKS!!!
   All other combinations of ints (e.g. 2, 3) of later major/minor versions Fails
   with the following console output msg:

   C:\Dev\Cherno\OpenGL\bin\Win32\Debug\OpenGL.exe (process 4936) exited with code -1.
   To automatically close the console when debugging stops,
   enable Tools->Options->Debugging->Automatically close the console when debugging stops.
   Press any key to close this window . . .
*/

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
glfwWindowHint(GLFW_OPENGL_ANY_PROFILE, GLFW_OPENGL_CORE_PROFILE);

/***** Create a windowed mode window and its OpenGL context
    glfwCreateWindow MUST BE PERFORMED BEFORE ANY glfwWindowHint(s) *****/

window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

if (window==NULL)
{
    return -1;
}

/** Make the window's context current - this MUST BE PERFORMED BEFORE glewInit() !!! */
glfwMakeContextCurrent(window);

/* Vid#11 (9:00) - should sync our Swap with the monitor's refresh rate
   and produce a smooth color change transition */

GLCall(glfwSwapInterval(1));

/** Vid#3: JT Added Modern OpenGL code here - MUST FOLLOW glfwMakeContextCurrent(window) */

if (glewInit() != GLEW_OK)
{
    std::cout << "glewInit() Error!" << std::endl;
}

/** Vid#3: JT Added Print Modern OpenGL Version code here */
std::cout << glGetString(GL_VERSION) << std::endl;

```

```

/* Vid#9B: Vertex Buffer - remove 2 duplicate vertices of the 6 vertices in position[] to implement
   an Index Buffer */
/* Create a New Scope {...} FROM Here TO BEFORE glfwTerminate(); */
{
    float positions[] = {
        -0.5f, -0.5f, 0.0f, // vertex 0
        0.5f, -0.5f, 0.0f, // vertex 1
        0.5f, 0.5f, 0.0f, // vertex 2
        -0.5f, 0.5f, 0.0f, // vertex 3
    };

    /* Vid9B: create Index Buffer using new indices[] array
       note: must be unsigned but can use char, short, int, etc. */

    unsigned int indices[] = {
        0, 1, 2, // 1st right triangle drawn CCW
        2, 3, 0 // 2nd inverted right triangle drawn CCW
    };

    /* Vid#14: (5:00) create VertexArray va & VertexBuffer vb AFTER creating Vertex Array Object (vao) */

    VertexArray va;

    VertexBuffer vb(positions, sizeof(positions));

    /* Vid#14: (24:00) create VertexBufferLayout */

    VertexBufferLayout layout;
    layout.Push<float>(3);
    va.AddBuffer(vb, layout);

    /* Vid#13: (19:35) Delete OR Comment Out IndexBuffer creation code,
       move to the new IndexBuffer Class, and replace with: */

    IndexBuffer ib(indices, 6);

    /* Vid#15: (12:15) Moved ParseShader to Shader.cpp constructor */
    /* Vid#15: (15:50) create Shader class instance and Bind it */

    Shader shader("res/shaders/Basic.shader");
    shader.Bind();

    /* Vid#15: (16:00) call shader.SetUniform4f("u_Color", 0.8f, 0.3f, 0.8f, 1.0f) */

    shader.SetUniform4f("u_Color", 0.8f, 0.3f, 0.8f, 1.0f);

```

```

/* Vid#15: (17:00) Unbind VertexArray, the Unbind Shader, Unbind VertexBuffer , and Unbind IndexBuffer
by setting each = 0 and re-bind all (3) inside the Rendering while loop before the glDraw cmd */

va.Unbind();

/* Vid#15: (16:50) delete glBindBuffers and add vb.UnBind() and ib.Unbind() */
//GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
//GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0));

vb.Unbind();
ib.Unbind();

/* Vid#15: (16:15) changed GLCall(glUseProgram(0) to shader.Unbind() */
shader.Unbind();

/* Vid#11 (8:00) - Animate Loop: 1st define 4 float variables: r, g, b, and i */

float r = 0.0f;           // red color float var initially set to zero
float increment = 0.05f;  // color animation float increment var initially set to 0.05

/* Games Render Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    /* Render here */
    GLCall(glClear(GL_COLOR_BUFFER_BIT));

    /* Vid#12: (4:45) Bind Shader (shader), Uniform (location), Vertex Buffer (buffer)
    and Index Buffer (ibo) BEFORE calling glDrawElements... */

    /* Vid#15: (16:20) changed GLCall(glUseProgram(shader) to shader.Bind() */
    shader.Bind();           // bind our shader

    /* Vid#15: (16:35) changed GLCall(glUniform4f(location, r, 0.3f, 0.8f, 1.0f))
    to shader.SetUniform4f("u_Color", r, 0.3f, 0.8f, 1.0f) */

    shader.SetUniform4f("u_Color", r, 0.3f, 0.8f, 1.0f);    // setup uniform(s)

    /* Vid#14: (24:00) Delete OR Comment out GLCall(glBindVertexArray(vao)); */
    //GLCall(glBindVertexArray(vao));    // bind our Vertex Array Object
    /* and Replace with va.Bind() Call */

    va.Bind();

    /* Vid#13: (20:00) delete OR comment out this glBindBuffer and replace with: */

    ib.Bind();           // new bind call to Index Buffer Class method

```

```

        GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr)); // Draw Elements call

        /* Vid#11 (8:30) check if r value > 1.0f --> set increment = -0.05f
           else if r value < 0.0f --> set increment = 0.05f */

        if (r > 1.0f)
            increment = -0.05f;
        else if (r < 0.0f)
            increment = 0.05f;

        r += increment;

        /* Swap front and back buffers */
        GLCall(glFWSwapBuffers(window));

        /* Poll for and process events */
        GLCall(glFWPollEvents());
    }

    /* Vid#15: (16:45) removed delete Shader b/c when the code hits end of scope,
       it will automatically be deleted by the ~destructor of the Shader class */
    //GLCall(glDeleteProgram(shader));
}

/* OpenGL GLFW Terminate destroys OpenGL context BEFORE IndexBuffer ~destructor is Called
   We now DO NOT have an OpenGL Context */
glfwTerminate();
return 0;
}

```

– **Test & Run (F5):(17:05) Failed!**

Severity	Code	Description	Project	File	Line	Suppression State
Error	C2374	'stream': redefinition; multiple initialization	OpenGL	C:\Dev\Cherno\OpenGL\src\Shader.cpp	167	

- **Root Cause:** Shader.cpp Shader::ParseShader method had duplicate lines of code: `std::ifstream stream(filepath);`
- **Re-Run (F5): Passed!** Draws Rectangle changing colors from pink to blue.

– **Fix Shader.cpp Shader::GetUniformLocation method**

- **Problem:** Every time we call Shader::SetUniform4f() it calls glUniform4f(GetUniformLocation(name), v0, v1, v2, v3), we retrieve the location again and again...
- **Solution:** Retrieve the location the 1st time and Cache that location thereafter.
 - **Shader.h modifications: (17:40) Add hash table with `#include <unordered_map>`**

➤ **Class Shader (17:50)**

```

/* Vid#15:(17:40) caching for UniformLocationCache */
std::unordered_map<std::string, int> m_UniformLocationCache;

```


...

- Shader.cpp modifications:

- unsigned int Shader::GetUniformLocation(const std::string& name) (18:00)

```
unsigned int Shader::GetUniformLocation(const std::string& name)
{
    /* Vid#15: (18:00) Check if UniformLocationCache contains the name
       and if it does, return the location, else GetUniformLocation() */

    if (m_UniformLocationCache.find(name) != m_UniformLocationCache.end())
        return m_UniformLocationCache[name];

    /* Vid#15: (13:45) should be int location like in Application.cpp */
    GLCall(int location = glGetUniformLocation(m_RendererID, name.c_str()));

    /* test for valid (-1) for Shader location when a Uniform is declared but not used yet */
    if (location == -1)
        std::cout << "Warning: uniform '" << name << "' doesn't exist!" << std::endl;

    /* cache the location improves the performance with multiple uniforms */
    m_UniformLocationCache[name] = location;

    /* return location */
    return location;
}
```

- Extending Uniform code: (18:00)

- Shader.h modifications

- Public: declarations

```
/* Vid#15: (20:00) Add Uniform1f declaration with string& name, float value */
void SetUniform1f(const std::string& name, float value);
```

- Shader.cpp modifications

- Copy Uniform4f method definition and Paste above as Uniform1f method definition

```
/* Vid#15: (20:15) Add Uniform1f definition with string& name, float value */
void Shader::SetUniform1f(const std::string& name, float value)
{
    /* Vid#15: (20:20) pass in location and value of the Uniform */
    GLCall(glUniform1f(GetUniformLocation(name), value));
}
```

- Next Step(s):

- Abstract the Renderer Class

Vid#16: Writing a Basic Renderer in OpenGL

▪ Concepts:

- How can we create a Fully Abstracted Renderer using a pipeline (glDraw calls)?
 - Using a Renderer API that we pass in the data and layout(s) to be Drawn allowing the Renderer to DO ALL THE WORK!
 - The Renderer is the central glue that ties everything together, can easily be debugged, and runs smoothly.
 - The Renderer is the factory where we pass in some parts, textures, layouts and request it draw something based on that input.
- Vid#16 Goal(s)
 - Take the last (2) remaining Application.cpp OpenGL Call (2:00) `GLCall(glClear(GL_COLOR_BUFFER_BIT));` and `GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr));` // Draw Elements call and taken care of by the Renderer.
 - Create a New Renderer Class and a new Draw function that takes in a group of arguments with the draw data and executes code to draw it on the screen.
 - The Renderer can be expanded to draw extremely complex and powerful graphic elements.

▪ Create New Renderer Class code: (3:10)

- Update Renderer: create New Renderer Class (3:20) – static (multiple instances) or singleton (one instance) ?
 - Update Header Renderer.h code: (3:30)
 - Includes: (5:20) `VertexArray.h`, `IndexBuffer.h`, and `Shader.h`
 - Add New Draw() method declaration that requires a `VertexArray`, `IndexBuffer`, and a valid `Shader` passed in as const references.
 - The `VertexArray` has the `VertexBuffer` bound to it.
 - The `IndexBuffer` has the index count and contains all (or part) of the Vertices to Draw

```
/* Vid#16 (5:00) add new Draw method declaration that takes in a VertexArray,
   IndexArray, and Shader references */

void Draw(const VertexArray& va, const IndexBuffer& ib, const Shader& shader) const;
```

- Update Implementation `Renderer.cpp` code: ()

- Add New Draw() method definition that takes in `VertexArray`, `IndexArray`, and `Shadow` references

```
#include "VertexArray.h"
#include "Renderer.h"

/* Vid#16: (9:45) #include VertexBufferLayout.h" - removes the infinite include looping error */
#include "VertexBufferLayout.h"

/* VertexArray constructor */
VertexArray::VertexArray()
{
    GLCall(glGenVertexArrays(1, &m_RendererID));
}
```

```

/* VertexArray ~destructor */
VertexArray::~VertexArray()
{
    GLCall(glDeleteVertexArrays(1, &m_RendererID));
}

/* Bind the Vertex Buffer and Setup the Vertex Layout */
void VertexArray::AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout)
{
    /* Bind the VertexArray */
    Bind();

    /* Bind the VertexBuffer */
    vb.Bind();

    /* Setup a Vertex Layout */
    const auto& elements = layout.GetElements();
    unsigned int offset = 0;

    for (unsigned int i = 0; i < elements.size(); i++)
    {
        const auto& element = elements[i];

        /* Enable or disable a generic vertex attribute array for index = 0 */
        GLCall(glEnableVertexAttribArray(i));

        /* When we specify the GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0));
        ,
        index 0 (1st param) of this Vertex Array is going to be bound to the currently bound
        glBindBuffer(GL_ARRAY_BUFFER, buffer).

        define an array of generic vertex attribute data
        1st param index = i,
        2nd param size = element.count for a (3) component vector that represents each Vertex position,
        3rd param symbolic constant = element.type,
        4th param normalized = converted directly as fixed-point values (GL_FALSE) 4th param,
        5th param stride = the amount of bytes between each Vertex based on
            2nd param vec2 (x, y, z) component position vector of 3 floats = 12 bytes,
        6th param pointer = position has an offset pointer = 0 */

        GLCall(glVertexAttribPointer(i, element.count, element.type,
            element.normalized, layout.GetStride(), (const void*)offset));

        offset += element.count * VertexBufferElement::GetSizeOfType(element.type);
    }
}

```

```

void VertexArray::Bind() const
{
    GLCall(glBindVertexArray(m_RendererID));
}

void VertexArray::Unbind() const
{
    GLCall(glBindVertexArray(0));
}

```

- Unbinding code in OpenGL is generally not required but helps with debugging code.
- **Modifications to VertexArray Class**
 - **Update Header VertexArray.h code: (5:10)**
 - **Delete or Comment out Includes: (9:35) VertexBufferLayout.h and add declaration of class VertexBufferLayout;**

```

#pragma once

#include "VertexBuffer.h"
/* Vid#16: (9:35) delete OR comment out #include "VertexBufferLayout.h" and
   instead declare class VertexBufferLayout; */
// #include "VertexBufferLayout.h"
class VertexBufferLayout;

class VertexArray
{
private:
    unsigned int m_RendererID;

public:
    /* VertexArray constructor */
    VertexArray();

    /* VertexArray ~destructor */
    ~VertexArray();

    /* AddBuffer takes in 2 const params:
       1st - VertexBuffer& vb,
       2nd - VertexBufferLayout &, layout */
    void AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout);

    /* VertexArray Bind method */
    void Bind() const;

    /* VertexArray Unbind method */
    void Unbind() const;
};

```

- Update Implementation VertexArray.cpp code: (5:10)

- Include VertexBufferLayout.h (9:45)

```
#include "VertexArray.h"
#include "Renderer.h"
/* Vid#16: (9:45) #include VertexBufferLayout.h" - removes the infinite include looping error */
#include "VertexBufferLayout.h"

VertexArray::VertexArray()
{
    GLCall(glGenVertexArrays(1, &m_RendererID));
}
VertexArray::~VertexArray()
{
    GLCall(glDeleteVertexArrays(1, &m_RendererID));
}
/* Bind the Vertex Buffer and Setup the Vertex Layout */
void VertexArray::AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout)
{
    /* Bind the VertexArray */
    Bind();

    /* Bind the VertexBuffer */
    vb.Bind();

    /* Setup a Vertex Layout */
    const auto& elements = layout.GetElements();
    unsigned int offset = 0;

    for (unsigned int i = 0; i < elements.size(); i++)
    {
        const auto& element = elements[i];

        /* Enable or disable a generic vertex attribute array for index = 0 */
        GLCall(glEnableVertexAttribArray(i));

        /* When we specify the GLCall(glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0));
        index 0 (1st param) of this Vertex Array is going to be bound to the currently bound
        glBindBuffer(GL_ARRAY_BUFFER, buffer).

        define an array of generic vertex attribute data
        1st param index = i,
        2nd param size = element.count for a (3) component vector that represents each Vertex position,
        3rd param symbolic constant = element.type,
        4th param normalized = converted directly as fixed-point values (GL_FALSE) 4th param,
        5th param stride = the amount of bytes between each Vertex based on
            2nd param vec2 (x, y, z) component position vector of 3 floats = 12 bytes,
        6th param pointer = position has an offset pointer = 0 */
```

```

        GLCall(glVertexAttribPointer(i, element.count, element.type,
            element.normalized, layout.GetStride(), (const void*)offset));

        offset += element.count * VertexBufferElement::GetSizeOfType(element.type);
    }
}

void VertexArray::Bind() const
{
    GLCall(glBindVertexArray(m_RendererID));
}

void VertexArray::Unbind() const
{
    GLCall(glBindVertexArray(0));
}

```

○ **Changes to Application.cpp (10:20)**

- **Include VertexBufferLayout.h** - removes the infinite include looping error
- **Create instance of Renderer class renderer**
- **Renderer.Draw()** call passing in: VertexArray va, IndexBuffer ib, and Shader shader
 - `renderer.Draw(va, ib, shader);`
- **Delete OR comment out glDrawElements b/c it's now inside the Renderer class**
 - `//GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr))`

```

#include <GL/glew.h> /* must be the 1st #include BEFORE ANY other #includes */
#include <GLFW/glfw3.h>
#include <iostream>

/* Vid#8 add includes to read, parse C++ external file: Basic.shader, and add to each Shader buffer */
#include <fstream>
#include <string>
#include <sstream>

/* Vid#13: (8:00) add include "Renderer.h" */
#include "Renderer.h"

/* Vid#13: (18:30) Add #includes for two new classes */
#include "VertexBuffer.h"
#include "IndexBuffer.h"

/* Vid#14: (23:50) Add #includes for new VertexArray class */
#include "VertexArray.h"

/* Vid#15: (15:40) Add #includes for new Shader class */
#include "Shader.h"

```

```

/* Vid#16: (10:20) #include VertexBufferLayout.h" - removes the infinite include looping error */
#include "VertexBufferLayout.h"

int main(void)
{
    GLFWwindow* window;

    /* Initialize the library */

    if (!glfwInit())
        return -1;

    /* Vid#12 (8:10) OpenGL GLFW Version 3.3 create an Open Context and Window with the Core Profile
    GLFW_OPENGL_CORE_PROFILE
    Note: ONLY (GLFW_CONTEXT_VERSION_MAJOR, 2) and (GLFW_CONTEXT_VERSION_MINOR, 1) WORKS!!!
    All other combinations of ints (e.g. 2, 3) of later major/minor versions Fails
    with the following console output msg:

    C:\Dev\Cherno\OpenGL\bin\Win32\Debug\OpenGL.exe (process 4936) exited with code -1.
    To automatically close the console when debugging stops,
    enable Tools->Options->Debugging->Automatically close the console when debugging stops.
    Press any key to close this window . . .

    */

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
    glfwWindowHint(GLFW_OPENGL_ANY_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    /***** Create a windowed mode window and its OpenGL context
    glfwCreateWindow MUST BE PERFORMED BEFORE ANY glfwWindowHint(s) *****/

    window = glfwCreateWindow(640, 480, "Hello World", NULL, NULL);

    if (window==NULL)
    {
        return -1;
    }

    /*** Make the window's context current - this MUST BE PERFORMED BEFORE glewInit() !!! ***/
    glfwMakeContextCurrent(window);

    /* Vid#11 (9:00) - should sync our Swap with the monitor's refresh rate
    and produce a smooth color change transition */

    GLCall(glfwSwapInterval(1));

```

```

/** Vid#3: JT Added Modern OpenGL code here - MUST FOLLOW glfwMakeContextCurrent(window) */

if (glewInit() != GLEW_OK)
{
    std::cout << "glewInit() Error!" << std::endl;
}

/** Vid#3: JT Added Print Modern OpenGL Version code here */

std::cout << glGetString(GL_VERSION) << std::endl;

/* Vid#9B: Vertex Buffer - remove 2 duplicate vertices of the 6 vertices in position[] to implement
an Index Buffer */

/* Create a New Scope {...} FROM Here TO BEFORE glfwTerminate(); */
{
    float positions[] = {
        -0.5f, -0.5f, 0.0f, // vertex 0
        0.5f, -0.5f, 0.0f, // vertex 1
        0.5f, 0.5f, 0.0f, // vertex 2
        -0.5f, 0.5f, 0.0f, // vertex 3
    };

    /* Vid9B: create Index Buffer using new indices[] array
    note: must be unsigned but can use char, short, int, etc. */

    unsigned int indices[] = {
        0, 1, 2, // 1st right triangle drawn CCW
        2, 3, 0 // 2nd inverted right triangle drawn CCW
    };

    /* Vid#14: (5:00) create VertexArray va and VertexBuffer vb AFTER creating Vertex Array Object */

    VertexArray va;

    //VertexBuffer vb(positions, 4 * 2 * sizeof(float));

    VertexBuffer vb(positions, sizeof(positions));

    /* Vid#14: (24:00) create VertexBufferLayout */

    VertexBufferLayout layout;

    layout.Push<float>(3);

    va.AddBuffer(vb, layout);

```



```

/* Vid#13: (19:35) Delete OR Comment Out IndexBuffer creation code,
   move to the new IndexBuffer Class, and replace with: */

IndexBuffer ib(indices, 6);

/* Vid#15: (12:15) Moved ParseShader to Shader.cpp constructor */

/* Vid#15: (15:50) create Shader class instance and Bind it */

Shader shader("res/shaders/Basic.shader");
shader.Bind();

/* Vid#15: (16:00) call shader.SetUniform4f("u_Color", 0.8f, 0.3f, 0.8f, 1.0f) */

shader.SetUniform4f("u_Color", 0.8f, 0.3f, 0.8f, 1.0f);

/* Vid#15: (17:00) Unbind VertexArray, the Unbind Shader, Unbind VertexBuffer , and Unbind IndexBuffer
by setting each = 0 and re-bind all (3) inside the Rendering while loop before the glDraw cmd */

va.Unbind();

/* Vid#15: (16:50) delete or comment out glBindBuffers and add vb.UnBind() and ib.Unbind() */
//GLCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
//GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0));

vb.Unbind();

ib.Unbind();

/* Vid#15: (16:15) changed GLCall(glUseProgram(0) to shader.Unbind() */

shader.Unbind();

/* Vid#16: (8:20) create instance of Renderer */

Renderer renderer;

/* Vid#11 (8:00) - Animate Loop: 1st define 4 float variables: r, g, b, and i */

float r = 0.0f;           // red color float var initially set to zero
float increment = 0.05f;  // color animation float increment var initially set to 0.05

/* Games Render Loop until the user closes the window */

while (!glfwWindowShouldClose(window))
{
    /* Vid#16: (10:30) Move glClear from Application.cpp to Renderer.h class Renderer public method
    and replace with renderer.Clear() Call */

```

```

//GLCall(glClearColor(GL_COLOR_BUFFER_BIT))

renderer.Clear();

/* Vid#12: (4:45) Bind Shader (shader), Uniform (location), Vertex Buffer (buffer)
   and Index Buffer (ibo) BEFORE calling glDrawElements... */

/* Vid#15: (16:20) changed GLCall(glUseProgram(shader) to shader.Bind() */

shader.Bind();      // bind our shader

/* Vid#15: (16:35) changed GLCall(glUniform4f(location, r, 0.3f, 0.8f, 1.0f))
   to hader.SetUniform4f("u_Color", r, 0.3f, 0.8f, 1.0f) */

shader.SetUniform4f("u_Color", r, 0.3f, 0.8f, 1.0f);    // setup uniform(s)

/* Vid#16: (8:30) renderer.Draw() call passing in:
   VertexArray va, IndexBuffer ib, and Shader shader */

renderer.Draw(va, ib, shader);

/* Vid#16: (8:35) delete OR comment out glDrawElements b/c it's now inside the Renderer class */
//GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr)); // Draw Elements call

/* Vid#11 (8:30) check if r value > 1.0f --> set increment = -0.05f
   else if r value < 0.0f --> set increment = 0.05f */

if (r > 1.0f)
    increment = -0.05f;
else if (r < 0.0f)
    increment = 0.05f;

r += increment;

/* Swap front and back buffers */

GLCall(GLFW_SWAP_BUFFER(window));

/* Poll for and process events */

GLCall(GLFW_POLL_EVENTS());
}

/* Vid#15: (16:45) removed delete Shader b/c when the code hits end of scope,
   it will automatically be deleted by the ~destructor of the Shader class */
//GLCall(glDeleteProgram(shader));
}

```

```

/* OpenGL GLFW Terminate destroys OpenGL context BEFORE IndexBuffer ~destructor is Called
   We now DO NOT have an OpenGL Context */

glfwTerminate();
return 0;
}

```

- **Cleanup Code:**

- **Application.cpp**

- **(10:30) Move glClear from Application.cpp to Renderer.h class Renderer public method and replace with renderer.Clear() Call**

- **Renderer.h**

- **(10:40) add new void Clear() declaration**

```
void Clear() const;
```

- **Renderer.cpp**

- **(10:45) Move glClear from Application.cpp to Renderer.h class Renderer public method**

```

/* Vid#16: (10:45) Move glClear from Application.cpp to Renderer.h class Renderer public method */
void Renderer::Clear() const
{
    GLCall(glClear(GL_COLOR_BUFFER_BIT));
}

```

- **Future : Renderer class should take in a VertexArray, an IndexBuffer, and a Material (instead of a Shader)**

- **A Material is a Shader plus a Set of Data and ALL of its Uniforms rendering specific Uniforms or per object uniform (e.g. Rectangle color like this application)**

- **Passing in a Material to the Renderer that Binds the Material which Binds the Shader and sets up all the Uniforms it needs to and then Call the Draw function**

- **Next Vid#17 Focus on Textures: Load, Render**

- **Later Focus on 3D Models with Debugging Tools, Lighting, GUI System, Text Rendering (write a library)**

Vid#17 Textures in OpenGL

▪ Concepts:

– What is a Texture?

- A graphical image attached to a surface used when Rendering some graphical elements on a screen.
- Create image file (photograph, photoshop, etc.) on the computer, upload the image to the GPU's memory, and use it in the Shader for a Drawing that Renders this Texture
- Using precalculated mathematical values baked into the Texture that are sampled by the Shader to perform cool lighting effects.

– Vid#17 Goal(s)

- Getting an Image from our computer onto a Surface in our OpenGL Application.
- Draw a Texture on the Rectangle instead of a solid color
 - How do we load a png file, sample it in a Shader, and Draw it on the screen?
 - Use stb_image Library (from GitHub;
https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa2dSeng4NFJNREN2S2M4OFc2SUQzejRJbjdBd3xBQ3Jtc0trb25mTkxNMzgxUV9EemV0N3haejVWY1hxeEF6RFhmc0RfMW1iVURDbHAyRVBBcGZtdmx5Q1dJTEYxRnVMcVdtYU1HSy1rQU52dWE1bIFHa2JyQU04eIRVWDgyOHBMQWIKOEtQa3BaelNGNDJlRjJFdw&q=https%3A%2F%2Fgithub.com%2Fnothings%2Fstb%2Fblob%2Fmaster%2Fstb_image.h) to Load the image (stb_image.png file) into the CPU memory
 - Give stb_image.png a filepath and it returns a pointer to an array buffer of RGBA pixel data.
 - Decode the png file (compression vs non-compressed) to get the raw Data
 - Upload the RGBA pixel array (buffer data) to the GPU through the OpenGL specification as a Texture
 - Bind the Texture when its time to Render
 - Modify the Shader to read that Texture when it's Drawing so the PixelShader (FragmentShader) reads from that Texture buffer memory and determine a color for each pixel per that Texture
 - Bind the modified Shader to the Texture slot.
 - Sample the Texture in the Shader
 - Execute glDrawElements to draw the sampled Texture on the screen.

▪ Create New code (4:45)

- Create new folder: textures in the res folder and store the Chernologo.png file there.
- Create new folder: vendor in the src folder
 - Create new folder: stb_image in the vendor folder
 - Create new item: Header File (.h) > stb_image.h > **Add**
 - stb_image.h is used to load the Chernologo.png using a filepath that returns a pointer to a buffer of RGBA bytes of pixels which gets uploaded to the GPU as a Texture
 - Next, we modify the VertexShader to read our Texture when it's Drawing
 - So the Fragment(Pixel)Shader reads from Texture memory and work out which pixels should be which color

- Open Browser Window and enter link: <https://github.com/nothings/stb> > CLK on stb_image.h
https://github.com/nothings/stb/blob/master/stb_image.h > CLK on Raw > Ctrl + a > Ctrl + c > Select stb_image.h file > Paste copied code

- Read stb_image.h Header File's Instructions

- #define STB_IMAGE_IMPLEMENTATION before you include this file in *one* C or C++ Implementation (stb_image.cpp) file to create the implementation.

- #define STB_IMAGE_IMPLEMENTATION

- #include "stb_image.h"

- Create new item in stb_image folder: (6:00) > C++ File (.cpp) > stb_image.cpp > **Add**

- Add code to stb_image.cpp

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_header.h"
```

- Compile: (6:30) Ctrl + F7 > Passed (Build: 1 Succeeded)

- Add New Class: Texture

- Add new item: (6:45) Header (.h) > Texture.h > **Add**

- Include "Texture.h"

- Create Texture Class with constructor, ~destructor, Bind(unsigned int slot = 0) and Unbind() methods and Refactor implementations to Texture.cpp

- Create Inline Helper methods: GetWidth() and GetHeight() and Refactor (option later) implementations to Texture.cpp

```
#pragma once

#include "Renderer.h"

class Texture
{
private:
    /* Renderer ID, filepath, and local buffer */
    unsigned int m_RendererID;
    std::string m_FilePath;
    unsigned char* m_LocalBuffer;

    /* Pixel Attributes width, height, and bits per pixel (BPP) */
    int m_Width, m_Height, m_BPP;

public:
    /* Texture constructor takes in a path */
    Texture(const std::string& path);
```

```

/* Texture ~destructor */
~Texture();

/* Vid#17: (8:10) Bind Texture FCN assigning Texture to Bind to slot 0
   provides ability to assign multiple Textures to different slots
   Desktops typically can have 32 slots, Mobile typically can support 8 slots
   Can ask OpenGL how many open slots available on current platform */

void Bind(unsigned int slot = 0) const;

/* Unbind Texture FCN */
void Unbind() const;

/* Vid#17: (7:45) Texture Helper FCN declarations - NOT USED IN THIS VIDEO! */
inline int GetWidth() const { return m_Width; }
inline int GetHeight() const { return m_Height; }

};

```

- **Add new item: (6:50) Source (.cpp) > Texture.cpp > **Add****

- **Include "Renderer.h"**

- **Initialize Constructor (9:30) :** m_RendererID(0), m_FilePath(path), m_LocalBuffer(nullptr), m_Width(0), m_Height(0), m_BPP(0)

```

#include <GL/glew.h> /* must be the 1st #include BEFORE ANY other #includes */
#include <GLFW/glfw3.h>

#include "Texture.h"

#include "vendor/stb_image/stb_image.h"

/* Vid#17 Texture Class */
Texture::Texture(const std::string& path)
    : m_RendererID(0), m_FilePath(path), m_LocalBuffer(nullptr), m_Width(0), m_Height(0), m_BPP(0)
{
    /* Load the image: Chernologo.png and flip it vertically on load,
       b/c OpenGL expects us to start at the bottom-left as 0,0 (not top-left) */
    stbi_set_flip_vertically_on_load(1);

    /* initialize m_LocalBuffer passing in:
       filename,
       (3) pointers to the width, height, and Bits Per Pixel of the file
       (Chernologo.png is 460x460 pixels), and (RGBA requires) 4 channels of variables */

    m_LocalBuffer = stbi_load(path.c_str(), &m_Width, &m_Height, &m_BPP, 4);
}

```

```

/* Load textures using ID and Bind 2D Texture */
GLCall(glGenTextures(1, &m_RendererID));
GLCall(glBindTexture(GL_TEXTURE_2D, m_RendererID));

/* Set (4) Required Texture Integer Parameters:
   (target, param_name: GL_TEXTURE_MIN_FILTER (resampled down per pixel), linear resampling)
   (target, param_name: GL_TEXTURE_MAG_FILTER (scale up render texture when area is larger in
pixels than the texture size), linear sampling)
   (target, param_name: GL_TEXTURE_WRAP_S (horizontal, x), clamp/not extend the area/no tiling)
   (target, param_name: GL_TEXTURE_WRAP_T (vertical, y), clamp/not extend the area/no tiling)
*/
GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR));
GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR));
GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));

/* Vid#17 (14:00) Send OpenGL the Data in m_LocalBuffer using glTexImage2D(
   1st param: GL_TEXTURE_2D - target,
   2nd param: 0 - single level = 0,
   3rd param: GL_RGBA8 - internal format is how OpenGL stores texture data,
   4th param: texture width,
   5th param: texture height,
   6th param: 0 pixel border,
   7th param: format of RGBA data,
   8th param: GL_UNSIGNED_BYTE - data type,
   9th param: pointer to pixels with data OR nullptr if just allocating space )
*/
GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, m_Width, m_Height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
m_LocalBuffer));
//glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, m_Width, m_Height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
m_LocalBuffer);

/* Unbind the Texture */
GLCall(glBindTexture(GL_TEXTURE_2D, 0));
//glBindTexture(GL_TEXTURE_2D, 0);

/* if m_LocalBuffer contains data, then free local buffer data */
if (m_LocalBuffer)
    stbi_image_free(m_LocalBuffer);
}

Texture::~Texture()
{
    /* destructor deletes texture(s) (passing in Renderer ID) from the GPU */
    GLCall(glDeleteTextures(1, &m_RendererID));
}

```

```

void Texture::Bind(unsigned int slot) const
{
    /* specify a texture 1st slot (GL_TEXTURE0) of 32 slots to activate
       BEFORE binding that texture to slot0 */
    GLCall(glActiveTexture(GL_TEXTURE0 + slot));
    GLCall(glBindTexture(GL_TEXTURE_2D, m_RendererID));
    //glActiveTexture(GL_TEXTURE0 + slot);
    //glBindTexture(GL_TEXTURE_2D, m_RendererID);
}

void Texture::Unbind() const
{
    /* Unbind the Texture */
    GLCall(glBindTexture(GL_TEXTURE_2D, 0));
}

```

- Add new code to Application.cpp (19:30)

```

/* Vid#17: (19:30) #include "Texture.h" */
#include "Texture.h"

...

/* Vid#17: (19:30) Create instance of Texture Class: texture with
   the path: "res/textures/ChernoLogo.png"
   then Bind() the texture with no param = 0 by default,
   */

    Texture texture("res/textures/ChernoLogo.png");
    texture.Bind();

/* Vid#17: (22:00) Call Shader SetUniformi passing in "u_Texture", bind to slot0*/

    shader.SetUniform1i("u_Texture", 0);

```

- Fix the Shader.h and Shader.cpp code:
 - /* Vid#17: (20:00) fix the Shader caching glGetUniformLocation FCN by changing unsigned int to int for testing location == -1 */
- Tell the Shader which Texture sampler (not integer) slot to sample from using a Uniform (20:50):
 - Send an integer Uniform to the Shader and that integer is the slot we bound a Texture to that we want to sample from
 - In the Shader code, we use that sampler integer passed in to sample that Texture and do what we want with it
 - Shader.h : Add an int Uniform declaration


```
/* Vid#17: (21:30) Add Uniform1i declaration with string& name, int value */
void SetUniform1i(const std::string& name, int value);
```
 - Shader.cpp: Add int Uniform definition


```
/* Vid#17: (21:45) Add Uniform1f definition with string& name, int value */
```



```
void Shader::SetUniform1i(const std::string& name, int value)
{
    /* Vid#17: (21:45) pass in location and value of the Uniform */
    GLCall(glUniform1i(GetUniformLocation(name), value));
}
```

- **Create Texture Coordinate System (22:30)** – to tell our geometry we are rendering which part of the texture to sample from
 - **PixelShader** (i.e. **FragmentShader**) goes through and rasterizes the **Rectangle** (i.e. goes through the **Rectangle**, determines the color, samples the texture, and draws every pixel according to its color and texture) starting from the bottom-left (0, 0) up to the top-right (1, 1) in **OpenGL**.
 - Need to specify for each vertex of the **Rectangle**, the area of the **Texture** that the **PixelShader** (i.e. **FragmentShader**) shall interpolate between the bottom-left (0, 0) and top-right (1, 1) so that if we are **Rendering** between 2 pixels, it will sample/pick the texture representing half-way between the 2 pixels.

- **Modify the Application.cpp (24:10)**

- **Vid#17: (24:10) float positions[]** : add another **vec2** to each vertices

```
/* Vid#17: (24:10) add another vec2 to each vertices */
float positions[] = {
    -0.5f, -0.5f, 0.0f, 0.0f,    // vertex 0 bottom-left vertex
    0.5f, -0.5f, 1.0f, 0.0f,    // vertex 1 right most edge of rectangle
    0.5f, 0.5f, 1.0f, 1.0f,    // vertex 2 top-right vertex
    -0.5f, 0.5f, 0.0f, 1.0f    // vertex 3 left most edge of rectangle
};
```

- **Vid#17: (24:50) add 2nd layout.Push**

```
// Vid#17 (25:05) VertexBuffer vb(positions, 4 * 4 * sizeof(float));
VertexBuffer vb(positions, sizeof(positions));

/* Vid#14: (24:00) create VertexBufferLayout */
VertexBufferLayout layout;
layout.Push<float>(2);

/* Vid#17: (24:50) add 2nd layout.Push<float>(0) */
layout.Push<float>(2);
va.AddBuffer(vb, layout);
```

- **Complete Re-Write of the Shader (Basic.shader) (25:15)**

- Existing Position coordinate: **layout(location = 0) vec4 position;**
- Add a new Texture coordinate: **layout(location = 1) vec2 texCoord;**
- Way to send data from the **VertexShader** to the **PixelShader** (i.e. **FragmentShader**) is through **Varying**
- **Run & Test (F5): (28:00)** Outputs **ChernLogo.png** BUT the resolution is terrible.
 - **Root Cause:** **Blending FCN HAS NOT been Setup AND NOT Enabled.**
 - **Solution:** **Enable Blender** (**GLCall(glEnable(GL_BLEND));**) and **Setup Blender** (**GLCall(glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA));**) in **Application.cpp** just below **unsigned int indices[]**

○ **Basic.shader file:**

```
// Vid#17: Basic.shader file modifications to support Texture: ChernLogo.png
#shader vertex
#version 330 core
in vec4 position;
//layout(location = 0) in vec4 position;

/* Vid#17: (25:15) Modify Basic.Shader to add new Texture coordinate */
in vec2 texCoord;
//layout(location = 1) in vec2 texCoord;

/* Vid#17: (26:30) method to output data in varying v_TexCoord
   from the VertexShader as input data to the FragmentShader */
out vec2 v_TexCoord;

void main()
{
    //gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
    gl_Position = position;

    /* Vid#17: (26:45) assign varying v_TexCoord with the texCoord input to the VertexShader */
    v_TexCoord = texCoord;
}

#shader fragment
#version 330 core

//layout(location = 0) out vec4 FragColor;
out vec4 color;

/* Vid#17: (26:50) method to output data in varying v_TexCoord
   from the VertexShader as input data to the FragmentShader */
in vec2 v_TexCoord;

/* Vid#17: add uniform for Color */
uniform vec4 u_Color;

/* Vid#17: (25:30) add uniform for Texture */
uniform sampler2D u_Texture;

void main()
{
    /* Vid#17: (26:00) assign texColor from texture */
    vec4 texColor = texture(u_Texture, v_TexCoord);

    /* Vid#17: (27:20) assign color from u_Color */
    color = texColor;
}
```

Vid#18 Blending in OpenGL

▪ Concepts:

- Review Vid#17's (2) Lines of Blender code in Application.cpp JUST BELOW `unsigned int indices[]`
 - Enable Blender (`GLCall(glEnable(GL_BLEND));`)
 - Setup Blender `GLCall(glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA));`
- What is Blending (1:25)?
 - How to Render something partially or fully Transparent (e.g. photoshop opacity less than 100%, glass, tinted glass, etc.)
 - In OpenGL, you MUST tell the GPU what, when, where, and how to do Transparency.
 - (3:50) Blending determines how we combine our output color (transparent blue square) with what is already in our target buffer (red square) forming a purple square of pixels that overlap.
 - Output = the (blue) color we output from our fragment shader (known as source)
 - Target buffer = the buffer our fragment shader is drawing to (known as destination)
 - Three ways to control blending?
 - `glEnable(GL_BLEND)` and `glDisable(GL_BLEND)` by default
 - `glBlendFunc(src, dest)` – how we blend the two colors together using specified src RGBA and dest RGBA factors
 - src – how the src RGBA factor is computed (default src multiplying factor is `GL_ONE` OR no change to src)
 - dest – how the dest RGBA factor is computed (default dest multiplying factor is `GL_ZERO` OR override it with src)
 - take each color channel of our src RGBA and dest RGB(A) and multiply those values with our respective specified src and dest RGBA factors.
 - `glBlendEquation(mode)`
 - mode – how we combine the src and dest colors
 - Default value is `GL_FUNC_ADD` – adds the src + dest colors together
 - ❖ (e.g. $\text{src} * \text{GL_ONE} + \text{dest} * \text{GL_ZERO} = \text{src value}$)
 - Vid#17 Example used:
 - `src = GL_SRC_ALPHA` – use the SRC ALPHA value (in this case, IT'S TRANSPARENT = 0)
 - `dest = GL_ONE_MINUS_SRC_ALPHA` OR INVERSE OF SRC_ALPHA ($1 - 0 = 1$)
 - If the `GL_SRC_ALPHA` channel value = 0 (completely Transparent, i.e. set to ZERO), the dest = ONE(1) – the SRC_ALPHA = ONE(1) OR the original src color.
 - Which means “use the destination color” – the color that's already in the buffer from the dest values. We don't use ANY of the src colors BECAUSE IT'S COMPLETELY TRANSPARENT (SET TO ZERO)
 - ❖ $R = (R_{\text{src}} * \text{SRC_ALPHA}(0)) + (R_{\text{dest}} * \text{ONE_MINUS_SRC_ALPHA}(1 - 0)) = R_{\text{dest}}$
 - ❖ $G = (G_{\text{src}} * \text{SRC_ALPHA}(0)) + (G_{\text{dest}} * \text{ONE_MINUS_SRC_ALPHA}(1 - 0)) = G_{\text{dest}}$
 - ❖ $B = (B_{\text{src}} * \text{SRC_ALPHA}(0)) + (B_{\text{dest}} * \text{ONE_MINUS_SRC_ALPHA}(1 - 0)) = B_{\text{dest}}$
 - ❖ $A = (A_{\text{src}} * \text{SRC_ALPHA}(0)) + (A_{\text{dest}} * \text{ONE_MINUS_SRC_ALPHA}(1 - 0)) = A_{\text{dest}}$
 - Vid#18 Example:
 - Our Source pixels are partially transparent white square, (1.0, 1.0, 1.0, 0.5) (RGBA), ALPHA = 0.5

- Destination Buffer channels are cleared to magenta, (1.0, 0.0, 1.0, 1.0) (RGBA), ALPHA = 1.0
- Using current Blending Settings:
 - ❖ $R = (R_{src}(1.0) * SRC_ALPHA(0.5)) + (R_{dest}(1.0) * ONE_MINUS_SRC_ALPHA(1 - 0.5)) = R_{dest}(1.0)$
 - ❖ $G = (G_{src}(1.0) * SRC_ALPHA(0.5)) + (G_{dest}(0.0) * ONE_MINUS_SRC_ALPHA(1 - 0.5)) = G_{dest}(0.5)$
 - ❖ $B = (B_{src}(1.0) * SRC_ALPHA(0.5)) + (B_{dest}(1.0) * ONE_MINUS_SRC_ALPHA(1 - 0.5)) = B_{dest}(1.0)$
 - ❖ $A = (A_{src}(0.5) * SRC_ALPHA(0.5)) + (A_{dest}(1.0) * ONE_MINUS_SRC_ALPHA(1 - 0.5)) = A_{dest}(0.75)$

▪ Concepts:

- How Math is used in graphics programming and how it's used in OpenGL?
 - Matrices – a Matrix is an Array of Numbers that can be manipulated by addition, subtraction, multiplication to get a new Set of Numbers used to position objects in a 3D world.
 - Vectors – are comprised of a Direction (i.e. angle) and a Magnitude (i.e. a length) on some 2D (x, y), 3D (x, y, z), and 4D (x, y, z, a) coordinate system that are generally used for Transformation.
 - Directional Vectors
 - Positional Vectors – in 2D (200, 100), 3D (200, 100, 50) or 4D (200, 100, 50, 75) space.
 - Transforms – are a way we can take VertexBuffer Matrix data and represent that data on the screen using position, direction, movement from a camera view (by moving the world around the object).
 - Position Vertices via Translation, Scaling, Rotation
 - Vid#18 Chernologo.png (460 x 460 pixel) Texture image drawn as a Blended Output to the 640 x 480 pixel window needs a Transformation to correct the Geometry aspect ratio in order to make the image appear square on our monitor.
- Using the Math(s) Library to correct the Geometry's aspect ratio (5:00):
 - Download Math Library (GLM) from GitHub: <https://github.com/g-truc/glm>
 - OpenGL Math Libraries – need to be column major ordering (requiring no re-ordering, no transposing, etc. already in the right format). Building a custom Math Library/API might use row major ordering.
 - Releases: 0.9.8.6 <https://github.com/g-truc/glm/tree/89e52e327d7a3ae61eb402850ba36ac4dd111987>
 - Code > Download Zip > glm-0.9.8.zip > Open file > Extract All > Browse > C:\Dev\Cherno\glm-0.9.8
 - Copy the glm folder and its contents in C:\Dev\Cherno\glm-0.9.8 to C:\Dev\Cherno\OpenGL\src\vendor
 - Use: (7:00) link to required .h header files (no .cpp files required)
 - VS2019: Configuration Properties > C++ > General > Additional Include Directories: Add > src/vendor;
 - In Texture.cpp, remove vendor in `#include "vendor/stb_image/stb_image.h"`
 - (7:30) OpenGL > Refresh project and verify glm folder is under src\vendor
 - (7:40) In Solutions Explorer, Select vendor folder > RT+CLK > Include in Project – allows searching glm functions, methods, etc.
 - (8:10) IMPORTANT: Select glm\detail folder and dummy.cpp file > RT+CLK > Exclude from Project otherwise its main() function will conflict with our main() function.
- Using a Projection Matrix (8:30) is a way we want to tell our Window how we want to Map ALL of our different Vertices to it.
 - Transform our VertexBuffer Vertices modeled in a 3D World for Drawing on a 2D 4:3 aspect ratio (640 x 480 pixel) Window Plane Surface
 - Application.cpp Updates: (9:30)
 - `#include "glm/glm.hpp"`
 - `#include "glm/gtc/matrix_transform.hpp"`

- (10:00) create 4x4 instance (proj) of an Orthographic Projection Matrix to map all of our vertex coordinates onto a 2D plane where objects that are further away DO NOT get smaller, 2D Rendering
 - `glm::highp_mat4 glm::ortho<float>(float left, float right, float bottom, float top, float zNear, float zFar)`
the absolute magnitude of left + right : bottom + top provides a 4:3 aspect ratio to fix our Vertex positioning using it in our Shader.
 - `glm::mat4 proj = glm::ortho(-2.0f, 2.0f, -1.5f, 1.5f, -1.0f, 1.0f);`
- (15:30) Run & Test (F5) – Passed! The Chernologo.png was correctly Transformed into a square logo on the 4:3 aspect ratio 2D window.
- Basic.shader Updates: (12:20)
 - Add mat4 Uniform to take in the Model View Projection (u_MVP) matrix from the CPU (Application.cpp) into this Shader - This example is just a Projection matrix (NO Model or View)
 - `uniform mat4 u_MVP;`
 - (13:00) multiply the Vertex position(s) with the new Projection Matrix (u_MVP), runs once per Vertex, and moves it into the appropriate space based on the new orthographic projection matrix
 - `gl_Position = u_MVP * position;`
 - (13:20) call shader matrix 4f passing in new Projection Matrix (u_MVP) and `glm::mat4 proj`
 - `shader.setUniformMat4f("u_MVP", proj);`
- Shader.h Updates: (13:30) – add new Uniform FCN
 - (13:40) Set Uniform Mat4f with string reference ptr to name and `const glm::mat4& matrix`
 - `void SetUniformMat4f(const std::string& name, const glm::mat4& matrix);`
 - (13:40) include `glm/glm.hpp`
 - `#include "glm/glm.hpp"`
 - (14:00) Refactor (RT+CLK) > SetUniformMat4f > Quick Actions and Refactorings > Create Definition of "SetUniformMat4f" in Shader.cpp

```
void Shader::SetUniformMat4f(const std::string& name, const glm::mat4& matrix)
{
    /* glUniformMatrix4fv
    1st param:  Get Uniform Location (name),
    2nd param:  count or # matrices,
    3rd param:  Transpose = GL_FALSE b/c we are using glm
    4th param:  pointer to the float array or memory address of matrix[col0][element0] */

    glUniformMatrix4fv(GetUniformLocation(name), 1, GL_FALSE, &matrix[0][0]);
}
```

▪ Concepts:

– What is Projection in OpenGL and Why do we use it?

- How we go from our 3D World Coordinate system to Map the Vertices to Project and Render them on our 2D Window computer screen's Surface.
- Projection – Maps our 3D Vertices to our 2D Window Surface
- Projection Matrix Transformations – converts the 3D Vertices into Normalized Device Coordinates that are Mapped to our 2D Window
- Normalized Space – is a coordinates system between -1 and +1 in x, y and z axis regardless of window resolution that ALL Vertices MUST fit inside to Rasterize onto the screen.
 - The left side of window is x-value = -1
 - The right side of window is x-value = +1
 - The bottom side of window y-value = -1
 - The top side of window y-value = +1
 - The z-value = +1 represents that maximum normalized (far) value, those z-value(s) end up being closer to the camera, indicating the object is at its maximum distance from the camera OR Smaller.
 - The z-value = -1 represents that minimum normalized (near) value, those z-value(s) end up being farther away from the camera, indicating the object is at its minimum distance from the camera OR Larger.
- Orthographic Projection – usually used for 2D Projection BUT can be used for 3D Projection in a Leveling Editor
- Perspective Projection – usually used for 3D Projection BUT can be use for 2D Projection in a 2D Platformer application
- Basic.shader Updates: (12:20)
 - (12:55) change FragmentShader's Chernologo.png to white rectangle
 - `color = vec4(1.0);`
 - (15:30) comment out for 3rd test to produce Chernologo.png towards bottom-left corner
 - `//color = vec4(1.0);`
- Application.cpp Updates: (13:00)
 - Change window size from 640x480 to 960x540
 - `window = glfwCreateWindow(960, 540, "Hello Projection Matrices", NULL, NULL);`
 - (13:30) hardcode test window pixel positions
 - `glm::mat4 proj = glm::ortho(0.0f, 960.0f, 0.0f, 540.0f, -1.0f, 1.0f);`
 - (14:15) hardcode test window pixel positions, Test & Run (F5): small white square corner bottom-left corner of window

```
float positions[] = {  
    -10.5f, -10.5f, 0.0f, 0.0f,    // vertex 0 bottom-left vertex  
     10.5f, -10.5f, 1.0f, 0.0f,    // vertex 1 right most edge of rectangle  
     10.5f, 10.5f, 1.0f, 1.0f,    // vertex 2 top-right vertex
```

```
-10.5f, 10.5f, 0.0f, 1.0f    // vertex 3 left most edge of rectangle  
};
```

- (15:00) hardcode 2nd test window pixel positions, Test & Run (F5): white square towards bottom-left corner of window

```
float positions[] = {  
    100.0f, 100.0f, 0.0f, 0.0f,    // vertex 0 bottom-left vertex  
    200.0f, 100.0f, 1.0f, 0.0f,    // vertex 1 right most edge of rectangle  
    200.0f, 200.0f, 1.0f, 1.0f,    // vertex 2 top-right vertex  
    100.0f, 200.0f, 0.0f, 1.0f    // vertex 3 left most edge of rectangle  
};
```

- (16:40) create glm::vec4 Vertex Position 100x100, 0x1
 - glm::vec4 vp(100.0f, 100.0f, 0.0f, 1.0f);
- (17:01) create glm::vec4 result = proj * vp, set breakpoint, Run & Test(F5) test 4, , (F11) and check result the original x=100, y=100 changed to fit within -1 to 1
 - glm::vec4 result = proj * vp;
 - x = -0.791666627, y = -0.629629612
- (18:00) Remove breakpoints(s), Run & Test(F5) shows the ChernLogo.png centered at x: -0.79, y: -0.63 from window center 0, 0 with x-min = -1, x-max = +1, y-min = -1, and y-max = +1.

Vid#21 Model View Projection (MVP) Matrices in OpenGL

▪ Concepts:

– Model View Projection (MVP) Matrix Pipeline:

- In Direct3D, Create the MVP by Multiplying a 4x4 Model Matrix times a 4x4 View Matrix times our 4x4 Projection Matrix (Vid#20).
- In OpenGL, Create the MVP by Multiplying our 4x4 Projection Matrix (Vid#20) times a 4x4 View Matrix times a 4x4 Model Matrix (Reverse Order)
 - The Projection Matrix – converts the 3D Vertices into Normalized Device Coordinates that are Mapped to our 2D (-1, +1, -1, +1, -1, +1) Window
 - The View Matrix (I Matrix) – is the way we simulate the View (translate position, change orientation) of our Camera by inversely transforming (translate, rotate, scale) our Object(s).
 - The Model Matrix – is the way we simulate and transform (translation, rotation, scale) the Vertex Model of our Object that we are Drawing.
 - The Resultant PVM Matrix is multiplied by our Vertex Position (Vertex Geometry stored in our Vertex Buffer)
- Application.cpp Updates: (8:00)
 - (8:10) add a View Matrix to position (translate/move, rotate/angle) our object(s) to the left effectively moving our camera to the right
 - `glm::mat4 view = glm::translate(glm::mat4(1.0f), glm::vec3(-100, 0, 0));`
 - (10:40) create MVP Matrix with Projection x View (No Model) only
 - `glm::mat4 mvp = proj * view;`
 - (10:50) change proj to mvp, Run & Test(F5): Passed, the Chernobyl.png moved 100 units to the left (x: -100)
 - `shader.SetUniformMat4f("u_MVP", mvp);`
 -

–

–