# The Cherno:  OpenGL

| | |
|---|---|
| **Engineer:** | **John Trites**<br>**January 7th, 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)

## 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, Vulcan, 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 ant 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 Cherno 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 Cherno OpenGL\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-vc201 → 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: 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 PEFORMED 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 glGenBuffer 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] verticies - positions by Alt+Shift Legacy vertices --> Ctrl+c**

```
float positions[6] = {
      -0.5f, -0.5f,
       0.0f,  0.5f,
       0.5f, -0.5f
};
```

      - **glGenBuffer(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 noramlized 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 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**
  - **Cherno's Shader code DOES NOT COMPILE! IT FAILS because of OpenGL Ver. 3.3 layout syntax in vertexShader and color syntax in fragmentShader:**
  - **Replaced Cherno vertexShader code: with LearnOpenGL Hello-Triangle code**
  - **Replace Cherno 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 glEnableVertexAttribArray(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, Invidia 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));
//(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" > Ok**
    - **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 PEFORMED 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(glfwSwapBuffers(window));

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

        /* 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;
}
```

- **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 must simpler than developing a game engine. You just right 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 defintion 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);

}
```

```cpp
/* 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 */

    GLCall(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;
}

/*** 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 PEFORMED 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 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#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**