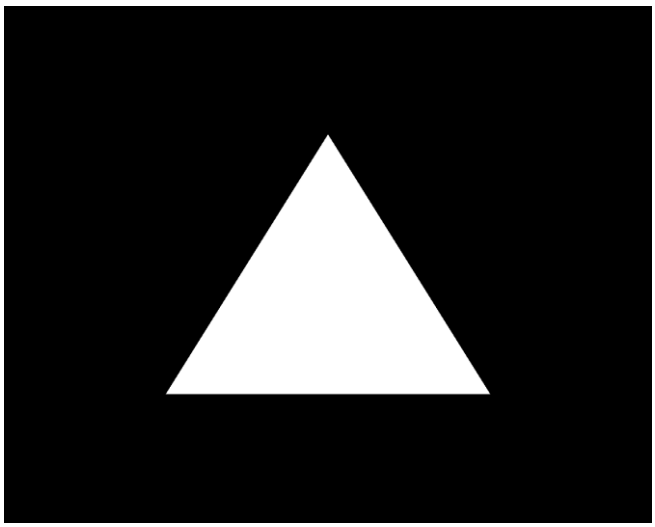
Tutorial 1: OpenGL 4 with SDL2

This tutorial covers how to create a simple OpenGL 4 render context and use it to output a single triangle.



This tutorial does not use any Operating System (OS) specific functions so the same code should work on any OpenGL/SDL supported platform. It can also be used in any C++ compiler or IDE suite that you may choose.

Part 1: Getting required libraries

Since OpenGL does not provide any windowing capabilities it relies on the host OS to create a window. Since this process is different on every different OS it is far simpler to use an existing windowing library that can handle the OS abstraction for us. There are many different available libraries that can be used to handle creating a window. In this tutorial SDL will be used as it supports all the main OS's as well as providing useful additional features such as keyboard/mouse/controller input, basic sound and others.

1. To start first download the SDL2 development libraries.
   https://www.libsdl.org/download-2.0.php

2. Extract the SDL2 include folder into a location accessible by your code. Also extract the "SDL2" library/binary (ignore SDLtest and SDLmain as they are not needed) and place that in an accessible location as well.

OpenGL has had many versions since its inception, each of these versions has added more available GL functions. Combine these with all the other extra vendor provided extensions to OpenGL and there are many possible functions that can be used. This can cause an issue as not every OS has up to date OpenGL available, although the GL functions may be supported by the graphics card and driver, they are not always directly exposed to a program. Windows is of particular note as it only provides OpenGL 1 support natively. In order to ensure that all OpenGL functions are available on all OS's the "GL Extension Wrangler" (GLEW) can be used to find and expose all available GL functions.

3. Download the GLEW binaries.
   http://glew.sourceforge.net/

GLEW provides 2 different library implementations; the first is the standard version while the second (postfixed with MX) provides support for multiple rendering contexts. Since for this and all future tutorials we will not be using multiple render contexts only the standard static “glew32s” library needs to be extracted.

4. Extract the GLEW include folder into a location accessible by your code (can be merged with the same include folder as SDL). Also extract the “glew32s” library.

Part 2: Creating a window

Once all the required libraries are downloaded the next step is to start writing the program.

5. To use both SDL and GLEW the appropriate header files need to be added to the source code. We define “GLEW_STATIC” to signal to GLEW that we are using the static version of the library. The program also needs an entry point which can take various forms depending on what OS is being targeted.

```
#define GLEW_STATIC
#include <GL/glew.h>
#include <SDL.h>
#ifdef _WIN32
#include <Windows.h>
#endif

#ifdef _WIN32
int WINAPI WinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE
hPrevInstance, _In_ LPSTR lpCmdLine, _In_ int)
#else
int main(int argc, char **argv)
#endif
```

In order to use SDL, each of its various subsystems need to be initialized. This is done through a call to “SDL_Init” and by passing a set of flags that specify which systems should be initialised. It should be noted that the event handling subsystem is automatically initialized when the video system is requested while the I/O and thread subsystems are always initialized by default.

6. For the moment only the video subsystem is required but more can be added at a later stage. As with all SDL functions, “SDL_Init” will return ‘0’ if everything succeeded. In the case that something fails we will use the supplied SDL logging functionality to output an error in an OS agnostic way.

```
// Initialize SDL
if (SDL_Init(SDL_INIT_VIDEO) != 0) {
    SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"Failed to initialize SDL: %s\n", SDL_GetError());
    return 1;
}
```

Before a SDL window can be created certain OpenGL parameters need to be set so that they are available during window creation. This requires setting the desired OpenGL version and compatibility profile. OpenGL 3.2 and up provides 2 different profiles; these are the core profile and compatibility profile. The core profile provides only those functions not marked as deprecated in the requested version of OpenGL. A compatibility profile provides all previous OpenGL functions.

7. For this tutorial the required OpenGL version will be set as 4.3. Setting this requires 2 calls to "SDL_GL_SetAttribute" in order to set both the major and minor OpenGL version tags before the window is created. Other versions can be used by simply replacing the specified version numbers. The profile will be set as core to prevent older deprecated OpenGL functions being used. After setting the GL version and profile we can now specify the back buffers to use. In this tutorial we will use double buffering with a 24bit z-buffer size.

```
// Use OpenGL 4.3 core profile
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK,
SDL_GL_CONTEXT_PROFILE_CORE);

// Turn on double buffering with a 24bit Z buffer
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);
```

8. Now we will create some variables used to store the desired window dimensions. These are created as simple global variables so they can easily be accessed by other function created in later tutorials while still being C compliant. However you are free to use some other method if you wish, global variables are just used here for simplicity.

```
// Declare window variables
int g_iWindowWidth = 1280;
int g_iWindowHeight = 1024;
bool g_bWindowFullscreen = false;
```

9. Next we can use SDL to create the required OpenGL window. This can be done using "SDL_CreateWindow" which takes a title for the new window, the x and y position to create the window at, the window width/height and various flags used to set the properties of the window. To automatically set the window position "SDL_WINDOWPOS_CENTERED" can be used to centre the window. We will also use "SDL_WINDOW_SHOWN" to ensure the window is visible at time of creation and "SDL_WINDOW_OPENGL" to ensure the created window is compatible with OpenGL. Additionally "SDL_WINDOW_FULLSCREEN" can be supplied to create a full screen window. Additionally, since SDL has been initialised at this point, should an error occur and the program terminate then we must ensure that SDL is correctly shutdown by calling "SDL_Quit".

```
// Create a SDL window
SDL_Window * Window = SDL_CreateWindow("AGT Tutorial",
SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, g_iWindowWidth,
g_iWindowHeight, SDL_WINDOW_SHOWN | SDL_WINDOW_OPENGL |
(g_bWindowFullscreen * SDL_WINDOW_FULLSCREEN));
if (Window == NULL) {
    SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"Failed to create OpenGL window: %s\n", SDL_GetError());
    SDL_Quit();
    return 1;
}
```

Once the SDL window is created an OpenGL context is required in order to use any OpenGL functions. Each OpenGL context is bound to the window it is created with and all OpenGL commands operate on the current context.

10. An OpenGL context can be created by SDL using the "SDL_GL_CreateContext" function. This function requires the input window that the context is bound to.

```
// Create a OpenGL Context
SDL_GLContext Context = SDL_GL_CreateContext(Window);
if (Context == NULL) {
    SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"Failed to create OpenGL context: %s\n", SDL_GetError());
    SDL_DestroyWindow(Window);
    SDL_Quit();
    return 1;
}
```

Vertical Synchronization (VSync) can be used to prevent frame tearing by ensuring that each buffer of the requested double-buffer is only presented to the screen during a "vertical refresh". This prevents updating the contents of the buffer being presented to the screen while it is still being used. This is the cause of frame tearing as the screen ends up displaying the first part of the original buffer contents and then the remainder of the screen gets replaced with the new buffer.

11. Enabling VSync can be performed using the "SDL_GL_SetSwapInterval" function. Passing in a '0' results in VSync being turned off. Passing a '1' turns VSync on and with platforms that support it passing a '-1' results in late swaps occurring immediately instead of waiting for the next refresh. This can be useful for preventing stuttering by keeping frame rates smoother.

```
// Enable VSync
SDL_GL_SetSwapInterval(-1);
```

12. Now that the window has been correctly created we need to setup any OpenGL resources that we wish to use. In this tutorial we will use a function called "GL_Init" that we will create in the next part of this tutorial.

```
//Initialize OpenGL
if (GL_Init()) {
    // *** Add message pump code here ***

    // Delete any created GL resources
    GL_Quit();
}
```

Now we need to create our program loop. This is the loop that will continually step over each frame and update any required data and then render the next frame. This requires the implementation of a message pump. An SDL program is constantly sent messages from internal subsystems and the operating system. Some of these messages are important and some are not but it is essential to handle the important messages otherwise the program will not function properly. The types of messages that are sent to the program range from things like keyboard or mouse input to specifics involving how a window is being rendered. It's through this message pump that SDL communicates with your program so important messages such as a quit command (someone presses the 'x' on the window or Alt+F4) are detected by the OS and sent to the program via a message. These messages could be sent at any time and many messages are constantly sent to the program.

13. A message pump is a programming method that will constantly check for any new message and if it detects any it will dispatch them to the appropriate function for action. SDL uses the "SDL_Event" object to store events. The SDL message pump works by creating an empty event object. We then use "SDL_PollEvent" to get any pending events. SDL will store any incoming events into a queue. Polling the queue will return the event at the front of the queue (the oldest event) and then remove it from the queue. If there are any events we then handle the ones we want to accordingly. For the moment we will just handle the "SDL_QUIT" event (generated when the OS requests the program terminates) and a single keyboard escape key press event. Both of these events will set the flag to instruct the program to quit. When there are no more events to handle the program then should call "GL_Render". This again is a custom function that we will add in the next section of this tutorial. This function will handle the OpenGL rendering operations. Lastly the program calls "SDL_GL_SwapWindow" which will cause the back-buffers to be swapped and the current one to be displayed to the screen.

```
// Start the program message pump
SDL_Event Event;
bool bQuit = false;
while (!bQuit) {
    // Poll SDL for buffered events
    while (SDL_PollEvent(&Event)) {
        if (Event.type == SDL_QUIT)
            bQuit = true;
        else if (Event.type == SDL_KEYDOWN) {
            if (Event.key.keysym.sym ==
SDLK_ESCAPE)
                bQuit = true;
        }
    }

    // Render the scene
    GL_Render();

    // Swap the back-buffer and present it
    SDL_GL_SwapWindow(Window);
}
```

Finally before exiting any initialized objects must be destroyed and SDL quit. This ensures all created objects are correctly destroyed. It is always good practice to clean-up any created data and so this should be performed at any point in code that causes the program to exit. This has been handled in all previously supplied code segments and should be remembered for any future modifications.

14. The provided SDL Destroy functions can be used to destroy any created SDL objects. Once they have been destroyed the initialised SDL context can be shutdown using a call to "SDL_Quit".

```
// Delete the OpenGL context, SDL window and shutdown SDL
SDL_GL_DeleteContext(Context);
SDL_DestroyWindow(Window);
SDL_Quit();
```

15. By creating blank "GL_Init", "GL_Render" and "GL_Quit" functions you should now be able to compile and run your program. At this point all we have done is create a blank window.

Part 3: OpenGL Rendering

Now that SDL has been used to set up a render window it is now time to use OpenGL to render something. In this tutorial we will start by simply rendering a basic triangle.

16. Create 3 variables used to store an OpenGL Vertex Array Object (VAO), a Vertex Buffer Object (VBO) and a Shader Program ID. In this tutorial these variables are made global so that "GL_Init", "GL_Quit" and "GL_Render" have access to them while being C compliant. However those using C++ could make them a member of a single class and then make the above functions members of that class if they wish.

```
// Declare OpenGL variables
GLuint g_uiVAO;
GLuint g_uiVBO;
GLuint g_uiMainProgram;
```

17. The "GL_Init" function is responsible for initialising any OpenGL variables that are required. To start with we will create this function and use it to initialise GLEW. GLEW uses function pointers to expose OpenGL functions. The GLEW initialisation routine must then be used to probe the OS and graphics driver to return all the available OpenGL functions. During the next few steps we will continue to add more code to the "GL_Init" function.

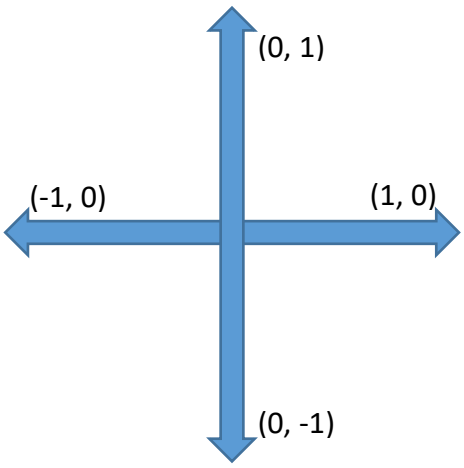
```
bool GL_Init()
{
    // Initialize GLEW
    glewExperimental = GL_TRUE; // Allow experimental extensions
    GLenum GlewError = glewInit();
    if (GlewError != GLEW_OK) {
        SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"Failed to initialize GLEW: %s\n", glewGetErrorString(GlewError));
        return false;
    }

    // *** Add remaining OpenGL initialisation here ***
}
```

18. Next we will set some initial OpenGL settings. The first setting uses "glClearColor" to set the colour that is applied to the back buffer when the contents are deleted. In this example we use the colour black (0,0,0). OpenGL colours use floating point values in the range (0.0->1.0) in (Red,Green,Blue) format. This means that a value of (1,1,1) is equivalent to white, (1,0,0) to red etc. Next "glCullFace" is used to specify the type of culling we would like to use. Triangle culling uses the direction that triangle vertices appear when projected on the screen to remove triangles. Where clockwise/counter-clockwise ordering is by default considered back/front facing respectively. This can improve performance by skipping triangles at the back of objects that won't be visible anyway. "glEnable" is used to enable the depth test using the z-buffer. This is required to sort the objects so that the closest objects always appear in front of those further away. Without z-buffering objects would appear in the order they are rendered. Finally we will also disable the stencil test for performance reasons as we won't be using this.

```
// Set up initial GL attributes
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
glEnable(GL_DEPTH_TEST);
glDisable(GL_STENCIL_TEST);
```

Since we are using OpenGL 4 we must use a programmable graphics pipeline. This means that we must write the shader code that controls how things are rendered. In this tutorial we will use a OpenGL vertex shader and a fragment shader. The vertex shader is responsible for creating the vertices in clip space and any associated data that we want to use in later shader stages. Each vertex shader is responsible for working on a single vertex and its data. This vertex data is then rasterised by the GPU. This process involves determining which pixels a given triangle covers (if any), interpolating any passed data as required for each covered pixel and then passing this information to the fragment shader. The fragment shader is then responsible for using this input information to determine the output (i.e. colour) for each required pixel.

A vertex shader must output a triangle vertex within clip space. OpenGL uses a right handed coordinate system where the x-axis points to the right, y-axis points upwards and the negative z-axis points into the screen. Normalized Clip space is then specified as the range (-1.0->1.0) in both the x, y and z directions. We will specify the triangle values of the VBO directly in this space. Where we will create a triangle with vertices at ((-0.5, -0.5, 0.0), (0.5, -0.5, 0.0), (0, 0.5, 0.0)) which will form an isosceles triangle centred at the middle of the screen.



19. Since the input triangle is already in normalized clip space all we have to do in the vertex shader is pass it out to the rasteriser. To create the new shader program we will write it as a string within the program that can be directly passed to OpenGL. The first line of the shader specifies the version of the OpenGL Shading Language (GLSL) that is required. All versions after OpenGL 3.3 just use the major and minor version numbers to determine the GLSL version. So since we are using OpenGL 4.3 we therefore have a version number of 430 (i.e. GL 3.3 will have a version of 330 etc.). We then specify we are only using core functionality by appending to the version string. Now the shader must specify the inputs (`in`) and outputs (`out`) that are passed to the shader program. In this example we have just a basic 2-dimensional input position specified in clip space. This is described as a single "`vec2`" that we can give any name we want (here we use "`v2VertexPos2D`"). OpenGL 3.3 and newer also allows for the "`layout(location)`" qualifier which makes things much simpler to access input data when there is more than one. Here we explicitly specify that "`v2VertexPos2D`" is found at location '`0`'. Next we specify the entry point of the shader. Much like a normal program OpenGL uses "`void main`" to specify the entry. All main has to do in this case is output the final vertex position. OpenGL has a mandatory inbuilt output variable called "`gl_Position`". This variable is a 4-dimensional variable so we must convert our 2-dimensional input into 4 dimensions. To do that we simply set the z value to '0' and as each vertex is a point in space we always set the w value to '1'.

```
// Create vertex shader source
const GLchar p_cVertexShaderSource[] = {
    "#version 430 core\n \
    layout(location = 0) in vec2 v2VertexPos2D;\n \
    void main() \n \
    { gl_Position = vec4(v2VertexPos2D, 0.0f, 1.0f); }"
};
```

20. Next the shader is loaded by OpenGL. Here we will create a convenience function called "GL_LoadShader" that we will create in a later step to handle loading any required shaders. This program takes as input a return to hold the shaders OpenGL ID, the type of the shader (here we use "GL_VERTEX_SHADER" to specify a vertex shader) and a char array holding the contents of the shader code.

```
// Create vertex shader
GLuint uiVertexShader;
if (!GL_LoadShader(uiVertexShader, GL_VERTEX_SHADER,
p_cVertexShaderSource))
    return false;
```

The fragment shader must output a colour value for each pixel it is run for. Unlike the vertex shader there isn't a built-in variable name that must be used to output the colour. However the first output is by default used to fill in the back buffers contents for the corresponding pixel. As this is a basic introduction to OpenGL we will just hardcode the output pixel value to a fixed colour.

21. The Fragment shader has no input data as it just simply outputs a hardcoded colour. In this case we will just use a value of (1, 1, 1) for white. We only have a single output variable (the colour) which we will give the name "v3ColourOut" (any name can be used). The "void main" function then simply outputs a hardcoded value for white. Once the fragment shader has been created we can then load it using the same "GL_LoadShader" function we used for the vertex shader, this time however we pass an input parameter of "GL_FRAGMENT_SHADER" to signal that this is a fragment shader.

```
// Create fragment shader source
const GLchar p_cFragmentShaderSource[] = {
    "#version 430 core\n \
    out vec3 v3FragOutput;\n \
    void main() \n \
    {\n \
        v3FragOutput = vec3(1.0f, 1.0f, 1.0f);\n \
    }"
};

// Create fragment shader
GLuint uiFragmentShader;
if (!GL_LoadShader(uiFragmentShader, GL_FRAGMENT_SHA
DER, p_cFragmentShaderSource))
    return false;
```

A basic OpenGL shader program requires both a vertex and a fragment shader combined. This step requires linking 2 separate shaders (vertex and fragment) to create the final program.

22. Just like for individual shader program loading the compete shader program will be linked using a convenience function called "GL_LoadShaders". We will create the contents of this function in a later step. The functions parameters include a return value to store the OpenGL ID of the program and the input OpenGL IDs of the already loaded vertex and fragment shaders.

```
// Create program
if (!GL_LoadShaders(g_uiMainProgram,
uiVertexShader, uiFragmentShader))
    return false;
```

23. Once the final OpenGL program has been linked the individual shaders are no longer required. So they can be deleted by using the "glDeleteShader" function and passing the IDs of each of the shaders.

```
// Clean up unneeded shaders
glDeleteShader(uiVertexShader);
glDeleteShader(uiFragmentShader);
```

OpenGL uses the concept of global state unlike some other APIs that use objects to store state. A side effect of this is that when a setting is changed that affects a certain type (triangle vertices for instance) it affects the global state and not just a specific instance. This makes it difficult to store different settings for different types (for instance you may want to render one object using a specified colour assigned to each vertex while another object uses a single colour value for the entire polygon). To facilitate storing different parameters together OpenGL has the concept of a Vertex Array Object (VAO). A VAO can be created and then set as the global state. Then after that any vertex object changes to the global state will affect the VAO and will be stored inside that VAO. Once setup VAOs can then be quickly swapped between to change multiple global geometry state parameters at once.

24. Create a VAO using "glGenVertexArrays". This takes as input a value specifying how many VAOs to create (here we only need 1) and a location to store the returned ID of the created VAO (here we use the VAO variable created earlier). We then use "glBindVertexArray" to bind the newly created VAO as the current one.

```
// Create a Vertex Array Object
glGenVertexArrays(1, &g_uiVAO);
glBindVertexArray(g_uiVAO);
```

OpenGL also has the concept of a Vertex Buffer Object (VBO). This is a section of memory that is ideally stored on the GPU that holds the polygon vertices to be rendered. This allows for many triangles to be stored in a single GPU buffer that can then be rendered using a single function call. This in theory allows for millions of triangles to be rendered at once.

25. Create a VBO using "glGenBuffers". This also takes an input specifying how many buffers to create as well as a return used to store the ID of the new VBO. "glBindBuffer" can then be used to bind the VBO as the current state. Here we bind it as "GL_ARRAY_BUFFER" type. This specifies that it should be bound as an array of vertices and any future function call that modifies the "GL_ARRAY_BUFFER" type will affect the currently bound VBO. Now the buffer is bound we can use "glBufferData" to copy some existing data into the attached buffer. The first parameter specifies that the buffer is the one attached to the "GL_ARRAY_BUFFER" mounting point, the second parameter specifies the size of the copy, the third specifies the source of the copy and the final parameter specifies the type of data in the buffer. Here we use "GL_STATIC_DRAW" to specify that the buffer is used for rendering and that it is static and won't be constantly updated. This allows the driver to determine the best location to place the data (since we are not updating the data the driver should place it into the GPU).

```
// Create VBO data
GLfloat fVertexData[] =
{
    -0.5f, -0.5f,
     0.5f, -0.5f,
     0.0f,  0.5f
};

// Create Vertex Buffer Object
glGenBuffers(1, &g_uiVBO);
glBindBuffer(GL_ARRAY_BUFFER, g_uiVBO);
glBufferData(GL_ARRAY_BUFFER,
sizeof(fVertexData), fVertexData,
GL_STATIC_DRAW);
```

Once the vertex data has been created we must now specify how that data is passed to the shader program. Each input in the vertex shader must have an associated data input in order to work. When using the "`layout(location)`" qualifier in the GLSL code then the location of the input data is already set. All that needs to be done is bind this location with the input buffer data.

26. In our basic shader there is only 1 input found in the vertex shader. This input takes vertex data directly from the VBO. To link the 2 together "`glVertexAttribPointer`" is used to specify which shader input corresponds to what section of data. Here the input parameters are the "`layout(location)`" ID (in this case '0'), the number of elements in each piece of data (here we need a "`vec2`" corresponding to 2 inputs) and the type of the inputs (here we have floating point values). The stride or distance between each element in the VBO (here it is the size of 2 floats. Note: stride is provided in case data is padded in the VBO in which case the stride would differ from the size of each element. In this example however that is not the case. Stride may also differ if multiple data elements are contained in the input as the stride value can be used to skip this information). The final input is the offset into the VBO to start reading data from (in this case it is '0' as we start from the beginning). Finally we enable the shader input using "`glEnableVertexAttribArray`".

```
// Specify location of data within buffer
glVertexAttribPointer(0, 2, GL_FLOAT,
GL_FALSE, 2 * sizeof(GLfloat),
(const GLvoid *)0);
glEnableVertexAttribArray(0);
```

27. Finally we bind the required shader program using "`glUseProgram`" and passing the OpenGL ID of the program we want to use. We do this in the "`GL_Init`" function as we only have 1 shader program that will remain constant for the entire program duration. Sometimes however more than 1 shader program will be used which will require setting and changing programs during rendering in which case "`glUseProgram`" would need to be moved to the "`GL_Render`" function.

```
// Specify program to use
glUseProgram(g_uiMainProgram);
```

This concludes the code required for the "`GL_Init`" function. The only missing components are the functions to load and link shader programs.

28. Loading a shader program requires several steps. The first is creating a new OpenGL ID for the shader program by using "`glCreateShader`". This function takes a single input specifying what type of shader to create ("`GL_VERTEX_SHADER`" for a vertex shader or "`GL_FRAGMENT_SHADER`" for a fragment shader). Once an ID has been acquired the shaders source code must be bound using "`glShaderSource`". This functions input parameters are the shader ID to link the source code with (here it is the ID returned by "`glCreateShader`"), the number of strings containing the shader source (in this case we just have 1 string), an array of strings containing the source code (since we have only 1 string we simply pass a pointer to the char array) and finally an array of string lengths (we don't have multiple strings so we just pass NULL). Once the shaders code is attached the shader must then be compiled using "`glCompileShader`". This will compile the shader code specifically for the GPU being used during runtime.

```
bool GL_LoadShader(GLuint & uiShader, GLenum ShaderType,
const GLchar * p_cShader)
{
    // Build and link the shader program
    uiShader = glCreateShader(ShaderType);
    glShaderSource(uiShader, 1, &p_cShader, NULL);
    glCompileShader(uiShader);

    // Check for errors
    GLint iTestReturn;
    glGetShaderiv(uiShader, GL_COMPILE_STATUS, &iTestReturn);
    if (iTestReturn == GL_FALSE) {
        GLchar p_cInfoLog[1024];
        int32_t iErrorLength;
        glGetShaderInfoLog(uiShader, 1024, &iErrorLength, p_cInfoLog);
        SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"Failed to compile shader: %s\n", p_cInfoLog);
        glDeleteShader(uiShader);
        return false;
    }
    return true;
}
```

After shader compilation we can use a call to "glGetShaderiv" specifying as input the shader ID being compiled, "GL_COMPILE_STATUS" in order to get the return code of the compilation status and a final parameter used to store the compilation result code. If the result is "GL_FALSE" then an error occurred during shader compilation. If for some reason there was an error "glGetShaderInfoLog" can be used to get the internal error information and print it to the log. This function has input parameters specifying the shader ID to get compile information for, the allocated size of the return buffer parameter, a return variable used to store how many characters were actually written to the return buffer parameter and finally a pointer to a pre-allocated char array return buffer. The returned error string can be of any unknown length up to 1024 characters. Therefore we pre-allocate a 1024 character char array to be used for the returned data where the actual length of the returned string is returned in the third parameter. Once we have the returned compile information we can output it to the log as usual. We should then clean-up the failed shader by deleting it and returning.

29. Once 2 shaders have been loaded they can be linked together into a complete shader program. This is similar to creating each individual shader. The first step is to create an OpenGL ID for the complete shader program using "glCreateProgram". Once the ID is created we can attach each of the individual shaders using the "glAttachShader" function. This function has 2 input parameters; the first is the ID of the shader program to attach to and the second is the ID of the shader to attach. Once all shaders are attached (here we only need to attach the vertex and fragment shader) the final program is then linked together using a call to "glLinkProgram". Just like with compiling shaders; "glGetProgramiv" can be used to get the status of the link operation this time with an input parameter of "GL_LINK_STATUS".

```
bool GL_LoadShaders(GLuint & uiShader, GLuint uiVertexShader,
GLuint uiFragmentShader)
{
    // Link the shaders
    uiShader = glCreateProgram();
    glAttachShader(uiShader, uiVertexShader);
    glAttachShader(uiShader, uiFragmentShader);
    glLinkProgram(uiShader);

    //Check for error in link
    GLint iTestReturn;
    glGetProgramiv(uiShader, GL_LINK_STATUS, &iTestReturn);
    if (iTestReturn == GL_FALSE) {
        GLchar p_cInfoLog[1024];
        int32_t iErrorLength;
        glGetShaderInfoLog(uiShader, 1024, &iErrorLength, p_cInfoLog);
        SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"Failed to link shaders: %s\n", p_cInfoLog);
        glDeleteProgram(uiShader);
        return false;
    }
    return true;
}
```

After any data has been created it is always good programming practice to ensure that it is properly destroyed in order to prevent memory leaks or other issues.

30. The "GL_Quit" function can now be created. This function is responsible for destroying any data that was created in "GL_Init" and still persists outside its scope. In this example that consists of the VAO, VBO and shader program. The shader program can be destroyed by using "glDeleteProgram" and passing the OpenGL ID of the program to destroy. The VBO can be destroyed by using "glDeleteBuffers". This function takes 2 inputs; the first being the number of VBO's to destroy (here we only have 1) and the second being an array of VBO IDs (as we only have 1 ID we simply pass a pointer to it – an array of 1 element). The VAO can be deleted using "glDeleteVertexArrays" which works identically to "glDeleteBuffers" except it takes the VAO ID as input.

```
void GL_Quit()
{
    // Release the shader program
    glDeleteProgram(g_uiMainProgram);

    // Delete VBO and VAO
    glDeleteBuffers(1, &g_uiVBO);
    glDeleteVertexArrays(1, &g_uiVAO);
}
```

Once all the required OpenGL values have been setup they can now be used for rendering. Since we are using a double buffered render context we have to ensure that each buffer is cleared before use. This is because the buffers are constantly being swapped with one buffer being displayed to the screen while the other buffer is being rendered into. Once rendering is complete the render buffer is then swapped with other buffer and now it becomes the buffer being displayed. The new render buffer however was previously the buffer being displayed and so it still contains the old information. Before it can be used again it should be cleared.

31. The "GL_Render" function can now be created. This function starts by just clearing the current render buffer using the "glClear" function. In order to correctly clear the render buffer we must clear the colour buffer by passing "GL_COLOR_BUFFER_BIT" and we must also clear the state of the z-buffer by passing "GL_DEPTH_BUFFER_BIT".

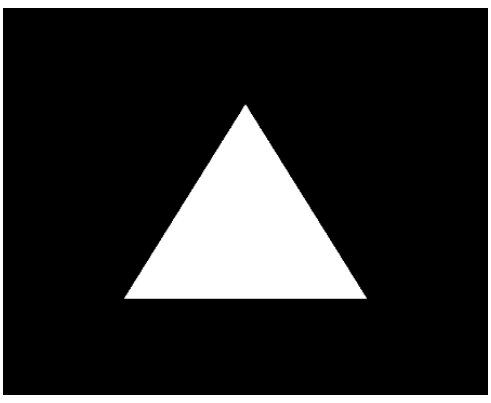
```
void GL_Render()
{
    // Clear the render output and depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // *** Add remaining render operations here ***
}
```

Since OpenGL uses global state we must modify this state with the appropriate values for each render call we perform. This requires first setting the shader program being used. Once a shader program is bound all draw calls occurring after it will all operate using the last previously set program. The same goes for any buffer settings such as the vertex attributes set using "glVertexAttribPointer". Therefore in order to draw the contents of a single buffer that buffer must first be bound and then the vertex attributes must be specified for how that buffer maps to the current bound program as well as any other required buffer attributes. This is where VAOs come in handy as they include all the required buffer state. So instead of needing to bind the buffers and reset each of their attributes we can simply bind the associated VAO and automatically set all of it.

32. Completing the render function requires first binding the VAO containing our buffer information using "glBindVertexArray" and passing the ID of the desired VAO. Finally, we can draw the contents of the buffer associated with the VAO using a "glDrawArrays" function call. The function has 2 inputs; the first specifies the type of data found in the vertex array (here we simply have a list of triangle vertices "GL_TRIANGLES". Note: there are other types that will be covered in a later tutorial), the second specifies the starting index (here we begin at the start by using index '0') and finally the last parameter specifies the number of elements to render (here we have 3 vertices in our buffer). The "glDrawArrays" function will then draw the data currently located in the bound vertex buffer attached to the "GL_ARRAY_BUFFER" mounting location. This mounting locations is one of the ones automatically set when binding a VAO.

```
// Specify VAO to use
glBindVertexArray(g_uiVAO);

// Draw the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);
```

33. The program is now complete and can be compiled and run. If everything has gone correctly you should see a single white triangle in the centre of the screen.



Part 4: Debug Call-backs

Debugging OpenGL programs can often be somewhat difficult as OpenGL functions don't actually return any sort of error values directly. This can make it easy to accidentally introduce an error into OpenGL code that may not become evident until later on in the program execution (such as when the complete frame is rendered). With more complex rendering operations it can become increasingly difficult to locate the exact source of a program error and then fix it. Traditional OpenGL provides a function "`glGetError`" that can be used to check the current global error state. This function returns the last set error value that has occurred in the program. This value does not necessarily represent the result of the last OpenGL function call but instead represents the last function call to generate an error (which may have been much earlier in code). Finding the exact location of an error in OpenGL code using "`glGetError`" often requires placing the function call after every OpenGL function call which significantly increases code size and decreases runtime performance.

OpenGL 4.3 introduced an alternate debugging option by introducing a call back system that can be registered with OpenGL and then allows for immediate notification of any errors at the exact point in code that the error occurred. Not only does this mechanism allow for more detailed error information but it can also be used to send notifications about potential performance improvements and other helpful information. It also only requires code to be added during OpenGL initialisation and not part of a programs inner loop which can help keep code a lot more readable while also allowing the call-back to be easily disabled.

34. To use the debug call-back mechanism new code needs to be added during OpenGL initialisation. This new code will be added to a new function called "`GLDebug_Init`" that we must then call within "`GL_Init`" just after GLEW has been initialised. We will also add this code so that it only takes effect when using debugging so that it does not impose a runtime penalty for release builds.

```
#ifdef _DEBUG
    // Initialise debug call-back
    GLDebug_Init();
#endif
```

35. Next we must ensure that when SDL creates the OpenGL window and associated context that it is done with debugging capabilities enabled. This is done by adding an extra function call within the main function just after we have set all the previous SDL attributes.

```
#ifdef _DEBUG
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_FLAGS, SDL_GL_CONTEXT_DEBUG_FLAG);
#endif
```

36. Now we need to add the definition of the "`GLDebug_Init`" function. This function will first enable synchronous call-backs using "`GL_DEBUG_OUTPUT_SYNCHRONOUS`"; synchronous call-backs can hurt performance but they allow for the call-back function to be immediately called when an issue occurs. Since we are only going to use call-backs when debugging then any performance hit is not much of an issue. Next we use "`glDebugMessageCallback`" to set our newly created call-back function as the one OpenGL should use for messaging. The second parameter to this function is for passing user defined data to the call-back (as we are not using this it is set to NULL). Lastly we can then use "`glDebugMessageControl`" to enable or disable specific types of call-back messages. The first use of this function is used to enable all possible call-back message types by passing the message type as "`GL_DONT_CARE`" and setting the last parameter to "`GL_TRUE`". We can then disable any

individual messages that we don't want by setting them to "GL_FALSE". To do this we perform an additional call to disable notification messages "GL_DEBUG_SEVERITY_NOTIFICATION" as these are usually just spam. Error messages come in with different severity levels so each of these can be explicitly disabled if desired by adding additional function calls with either "GL_DEBUG_SEVERITY_LOW", "GL_DEBUG_SEVERITY_MEDIUM" or "GL_DEBUG_SEVERITY_HIGH" set accordingly.

```
void GLDebug_Init()
{
    //Allow for synchronous callbacks.
    glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);

    //Set up the debug info callback
    glDebugMessageCallback((GLDEBUGPROC)&DebugCallback, NULL);

    //Set up the type of debug information we want to receive
    uint32_t uiUnusedIDs = 0;
    glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0,
&uiUnusedIDs, GL_TRUE); //Enable all
    glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE,
GL_DEBUG_SEVERITY_NOTIFICATION, 0, NULL, GL_FALSE); //Disable notifications
}
```

37. Now we need to define the actual function that we want to be called back by OpenGL. This function must take a set of specific inputs with 4 input integers that define the call-backs source, type, ID and severity. It also passes back a string containing a user readable message, along with the messages length. Lastly it returns a user pointer which can be used to pass any user defined data in and out of the call-back function. For this tutorial we will not be using this feature so it will be set to NULL. The call-back function will then convert the input integers to user readable strings and output them along with the message using SDLs inbuilt logging functionality.

```
static char * gp_cSeverity[] = {"High", "Medium", "Low", "Notification"};
static char * gp_cType[] = {"Error", "Deprecated", "Undefined",
"Portability", "Performance", "Other"};
static char * gp_cSource[] = {"OpenGL", "OS", "GLSL Compiler",
"3rd Party", "Application", "Other"};

void APIENTRY DebugCallback(uint32_t uiSource, uint32_t uiType,
uint32_t uiID, uint32_t uiSeverity, int32_t iLength,
const char * p_cMessage, void* p_UserParam)
{
    // *** Add code to convert inputs to strings here ***

    // Output to the Log
    SDL_LogCritical(SDL_LOG_CATEGORY_APPLICATION,
"OpenGL Debug: Severity=%s, Type=%s, Source=%s - %s", gp_cSeverity[uiSevID],
gp_cType[uiTypeID], gp_cSource[uiSourceID], p_cMessage);
    if (uiSeverity == GL_DEBUG_SEVERITY_HIGH) {
        //This a serious error so we need to shutdown the program
        SDL_Event event;
        event.type = SDL_QUIT;
        SDL_PushEvent(&event);
    }
}
```

38. To convert the relevant input integers to output strings we have created 3 string arrays so all that needs to be done is to convert the corresponding integer to an index into the required string array. The following code show how to convert the severity values to their corresponding array indexes.

```
// Get the severity
uint32_t uiSevID = 3;
switch (uiSeverity) {
    case GL_DEBUG_SEVERITY_HIGH:
        uiSevID = 0; break;
    case GL_DEBUG_SEVERITY_MEDIUM:
        uiSevID = 1; break;
    case GL_DEBUG_SEVERITY_LOW:
        uiSevID = 2; break;
    case GL_DEBUG_SEVERITY_NOTIFICATION:
    default:
        uiSevID = 3; break;
}
// Get the type
uint32_t uiTypeID = 5;
switch (uiType) {
    case GL_DEBUG_TYPE_ERROR:
        uiTypeID = 0; break;
    case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR:
        uiTypeID = 1; break;
    case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR:
        uiTypeID = 2; break;
    case GL_DEBUG_TYPE_PORTABILITY:
        uiTypeID = 3; break;
    case GL_DEBUG_TYPE_PERFORMANCE:
        uiTypeID = 4; break;
    case GL_DEBUG_TYPE_OTHER:
    default:
        uiTypeID = 5; break;
}
// Get the source
uint32_t uiSourceID = 5;
switch (uiSource) {
    case GL_DEBUG_SOURCE_API:
        uiSourceID = 0; break;
    case GL_DEBUG_SOURCE_WINDOW_SYSTEM:
        uiSourceID = 1; break;
    case GL_DEBUG_SOURCE_SHADER_COMPILER:
        uiSourceID = 2; break;
    case GL_DEBUG_SOURCE_THIRD_PARTY:
        uiSourceID = 3; break;
    case GL_DEBUG_SOURCE_APPLICATION:
        uiSourceID = 4; break;
    case GL_DEBUG_SOURCE_OTHER:
    default:
        uiSourceID = 5; break;
}
```

39. This completes the code necessary for debug call-backs. At this point there should be no noticeable difference as the debugging functionality is only useful when an error occurs. However, this may become much more useful for later tutorials.

Part 5: Extra

40. Now that the program is complete try and modify it so that the output triangle is yellow on a blue background.

41. Now modify the vertex buffer so that the triangle is upside down

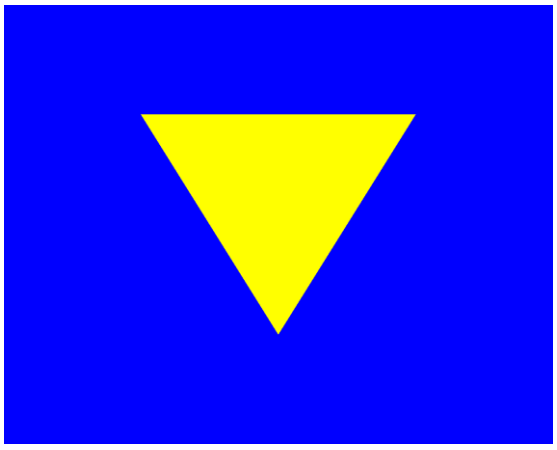
42. Next set the program to run in full screen mode.

43. Currently the programs resolution is hardcoded. It would be nicer if it automatically detected the current running resolution of the host computer and set itself to use that. The current desktop resolution can be determined by using a call to SDLs "`SDL_GetCurrentDisplayMode`". This returns the current screen width and height in "`CurrentDisplay.w`" and "`CurrentDisplay.h`" respectively for the specified screen (here we default to '0' to specify the first screen).

```
//Get desktop resolution
SDL_DisplayMode CurrentDisplay;
SDL_GetCurrentDisplayMode(0, &CurrentDisplay);
```

    Use these to automatically set the screen to the desktop resolution only when in full screen. To do this use 2 globally accessible variables to hold the used window width/height instead of using the constants as done previously.

44. Creating OpenGL shaders directly in a string is cumbersome and hard to read. A cleaner way would be to create a separate ".glsl" shader file for each shader and then read it into a string instead. Modify your program so that the shader code is in separate ".glsl" files that are loaded from file.



Note: Using with older OpenGL versions

This tutorial uses OpenGL functionality that is not found in older versions. The debug call-back functionality requires OpenGL 4.3 so in order to run this tutorial with older OpenGL versions it would be necessary to disable the use of the debug call-back. This can be achieved by simply commenting out the "`GLDebug_Init`" function call in "`GL_Init`".

This tutorial can then be run with older OpenGL by simply changing the requested OpenGL version to a lower supported one. For instance, this tutorial will run with OpenGL 3.3 by simply changing the SDL code that requests an OpenGL context. The current code to request an OpenGL version is:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 4);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
```

This can be converted to use OpenGL 3.3 instead by using:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
```

Layout qualifiers are also used in this tutorial to specify the location of vertex shader inputs. This functionality was added in OpenGL 3.3 and must be removed in order to support older versions. For instace the current vertex shader code is:

```
#version 430 core

layout( location = 0 ) in vec2 v2VertexPos2D;

void main() {
    gl_Position = vec4( v2VertexPos2D, 0.0f, 1.0f );
}
```

This can be made compatible with older versions by dropping the use of "`layout(location)`" and adding host code to setup the location explicitly. It is also required to change the requested shader version (here we use "140" which is the equivalent for version OpenGL 3.1).

```
#version 140

in vec2 v2VertexPos2D;

void main() {
    gl_Position = vec4( v2VertexPos2D, 0.0f, 1.0f );
}
```

The required host code must be added after the shader program has been created using "`glCreateProgram`" but before the shader is linked using "`glLinkProgram`". This code will then use "`glBindAttribLocation`" to bind the shader variable specified by name (here "`v2VertexPos2D`") to the requested location (here it is '0' to match the replaced use of "`layout( location )`").

```
// Specify vertex input location
glBindAttribLocation(g_uiMainProgram, 0, "v2VertexPos2D");
```