My First Triangle

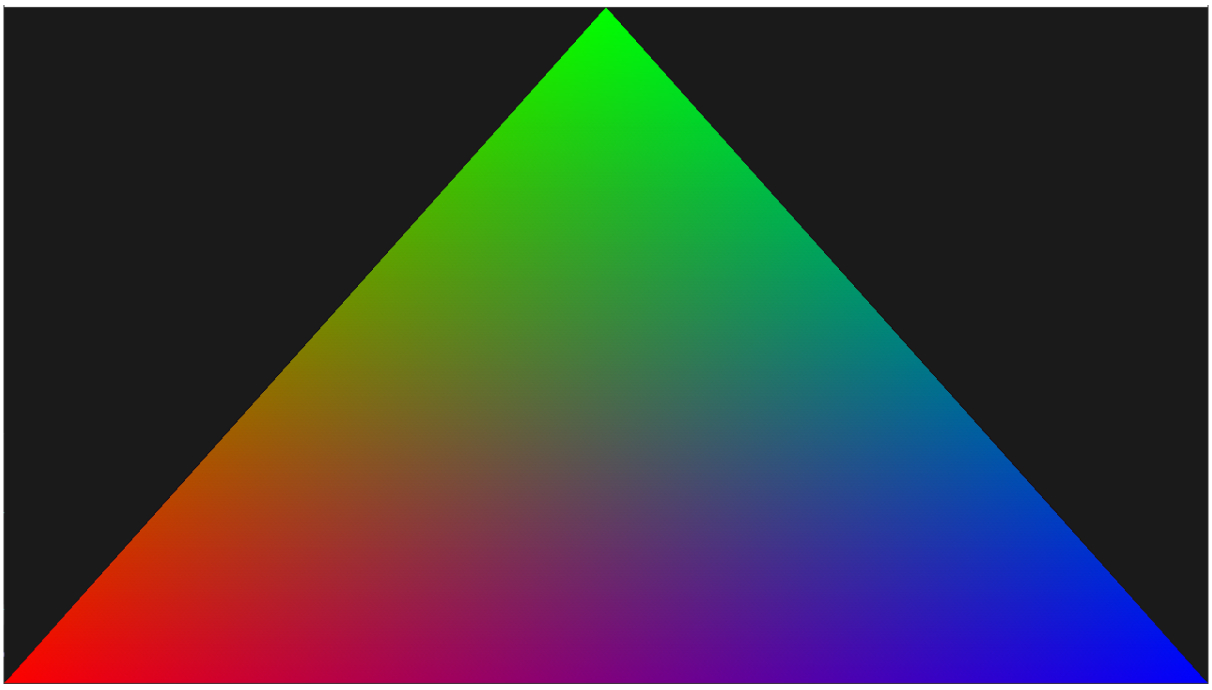
In the Skateboard engine

The purpose of this document is to help you getting started with programming graphics in the Skateboard Engine. We will start from a fresh, empty project where the bare minimum has already been set up to run a graphics application on Windows. By the end of this exercise, you should be able to render a triangle in a 3D world with a moving camera.

Task breakdown:

- Getting started and understanding the foundation of the Skateboard engine
- Create a vertex buffer and an index buffer to represent the triangle
- Create a rasterization pipeline & draw the triangle
    - Includes the creation of a vertex shader and pixel shader to render our triangle on the GPU
- Create a perspective camera (not from scratch, the engine already has one!) and supply its corresponding matrices to the shaders
- Create a basic debugging user interface
- (Bonus) Switching the platform to Playstation 5

Note that if you struggle to follow along, our full code snippets resulting from this exercise can be found in appendix A, B, C, D, E & F.



Foreword

The base of the engine was created following **The Cherno**’s game engine series, as it has been the first time that I, a recent graduate and original coder of this little engine, learned how to construct a proper engine from scratch. Similarities are mostly found on the original layout of the projects and the utmost basic concepts, but will differ when it comes to graphics and operating system abstractions. This also means that mistakes, errors and badly designed code are present in the *Skateboard* engine and you should not consider it as industry standards, but rather as a tool to help you understand these concepts earlier on! I heavily recommend watching and learning from their material as soon as possible if you are interested in engine development. Along with the *Skateboard* engine, this should give you a good amount of material to create an engine of your own, better and stronger than this one (:

Getting Started

- Make sure to clone from the skateboard engine repository.
    - This will also clone dependencies, such as ImGui, GLM, or SpdLog
- Open the Visual Studio Solution *Skateboard.sln* and compile. If everything went right there should be no errors.
- We strongly recommend enabling *Show All Files* on your Visual Studio projects (if not enabled already) as it will allow you to browse and structure the project with real file paths and directory. That is, if a file is shown to be under *src > Skateboard > Camera.cpp*, then that is the same directory as it is on your disk.

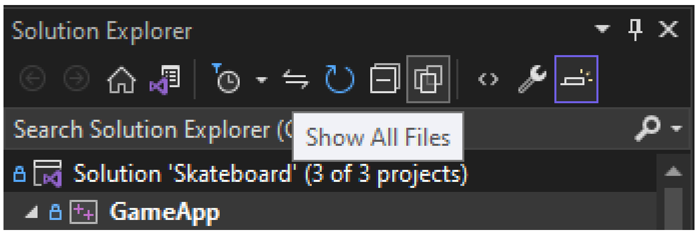
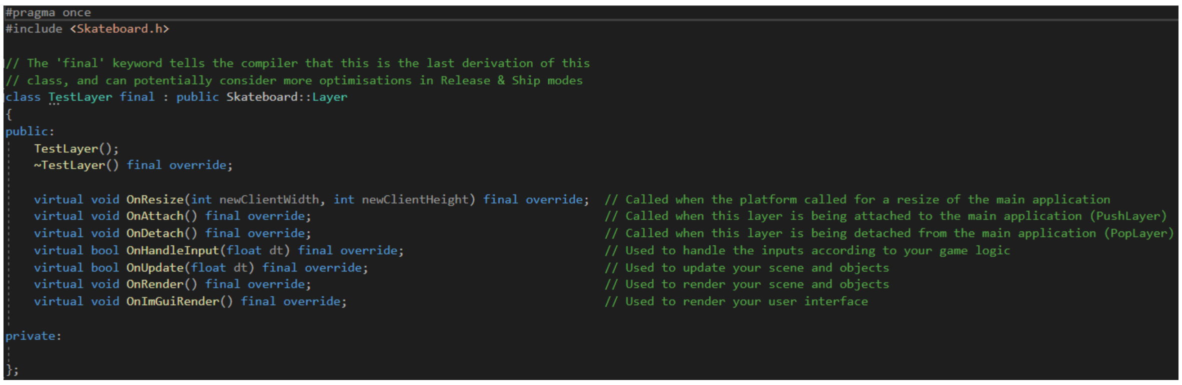

Solution Explorer
Search Solution Explorer (C
Show All Files
Solution 'Skateboard' (3 of 3 projects)
GameApp

Figure 1 - Showing All Files on a project

The ***GameApp* project** is where you will be mostly working. The engine code comes from the ***Skateboard* project**, which is compiled as a static library and linked to your game application on compilation. As we'll see in the last step, *Skateboard* is a cross-platform engine, which means it can run on different platforms by using methods of abstraction for graphics and operating system concepts. In the ***Skateboard* project**, you can find the different platform implementations under *src > Platform > …*, where *DirectX 12* & *Windows* are the graphics and operating system implementations for Windows respectively, as well as *AGC* & *Playstation* being the Playstation 5 equivalents. If you are interested in looking at how these concepts are implemented on the different platforms, feel free to have a look at these files in your own time! It is not the current focus of this course, but we will cover a bit more about the Playstation 5 side of things at the end of this document and any tutor will be happy to answer your questions.
Now have a look at the *GameApp.cpp* file located in the ***GameApp* project**. This file defines how you create the game application. For now, it is relatively empty and it will be up to you to change it to your goals. If you go to the declaration of *Skateboard::CreateApplication* (click on the function and press CTRL + F12), you will see that it is being referenced in two places: once in the base application class, and once as an external in the entry point. This design allows you to tell the engine (i.e. the static library) that this function will exist upon final linking but doesn't exist in this scope yet, and therefore to let you (the user) decide how to create an application instance from the *GameApp* project instead of inside the engine. In this case, we are just creating a simple, empty application. So let's go ahead and try to add a first layer of functionality to our game! We will tackle this by introducing the concept of game **layers and overlays**.
To make things very simple, you can only create one application (**Singleton** pattern). But what if you wanted to have multiple windows, overlays, etc? Well you will need to use the layers and overlays interfaces provided by the *Skateboard* engine. Essentially, your application can consider one or multiple layers to be executed at runtime (exercise: locate and have a look at the Application::Run() function). You can think of a layer as its own

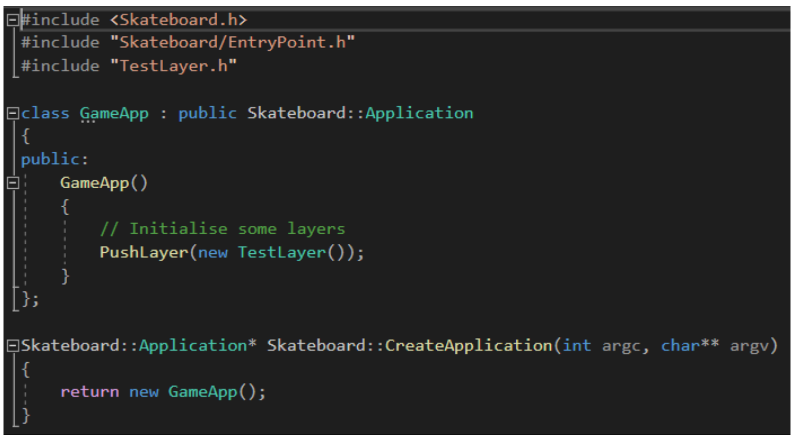
separate little application, where you will handle some inputs, update and draw things if necessary. This allows you to consider multiple tasks separately, such as having a layer for a mesh editing window and another for a real-time game view of your editor. Layers are pushed on a layer stack using the *PushLayer()* function. Note that layers will be added in order as you push them. Overlays are using the exact same interface as layers, but they will always be added on the top of the stack (above all layers) using the *PushOverlay()* function. This is useful when creating user-interface only layers.
Alright, that was a lot of rambling, now let's get started for real. Go ahead and create a new class in the source files of your ***GameApp* project** (Right click on the '*src*' folder in the project > Add > New Item… > Create a header then create a cpp). As we will create a layer, we named our files *TestLayer.h* and *TestLayer.cpp*, but feel free to choose a more sensible name for your layer. In the header file, you will need to include the engine using *#include <Skateboard.h>*. In the CPP file, make sure to include *TestLayer.h* (or the name of your layer). We can now start to create our layer class by using inheritance on the *Skateboard::Layer* object. See *Figure 2* for our implementation. Note that you will not necessarily need to override all the virtual functions and you should feel free to only consider the ones you need in your layer later on. However, for this example, we will just leave them all empty. Note that the *HandleInputs* and *Update* functions must return **false** when you wish to **exit** the application completely, so make sure to return **true** as a start and in every other situation.



```cpp
#pragma once
#include <Skateboard.h>

// The 'final' keyword tells the compiler that this is the last derivation of this
// class, and can potentially consider more optimisations in Release & Ship modes
class TestLayer final : public Skateboard::Layer
{
public:
    TestLayer();
    ~TestLayer() final override;

    virtual void OnResize(int newClientWidth, int newClientHeight) final override;  // Called when the platform called for a resize of the main application
    virtual void OnAttach() final override;                                          // Called when this layer is being attached to the main application (PushLayer)
    virtual void OnDetach() final override;                                          // Called when this layer is being detached from the main application (PopLayer)
    virtual bool OnHandleInput(float dt) final override;                             // Used to handle the inputs according to your game logic
    virtual bool OnUpdate(float dt) final override;                                  // Used to update your scene and objects
    virtual void OnRender() final override;                                          // Used to render your scene and objects
    virtual void OnImGuiRender() final override;                                     // Used to render your user interface

private:

};
```

Figure 2 - Our implementation of the TestLayer class in the header file by overriding all the virtual functions

Now, before getting started with graphics, the last step is simply to push this layer to the application by including our layer in the *GameApp.cpp* file and using the *PushLayer* function. See *Figure 3*, and note that ownership of the pointer is being transferred to the layer stack (i.e. the layer stack will call 'delete' on exit)!



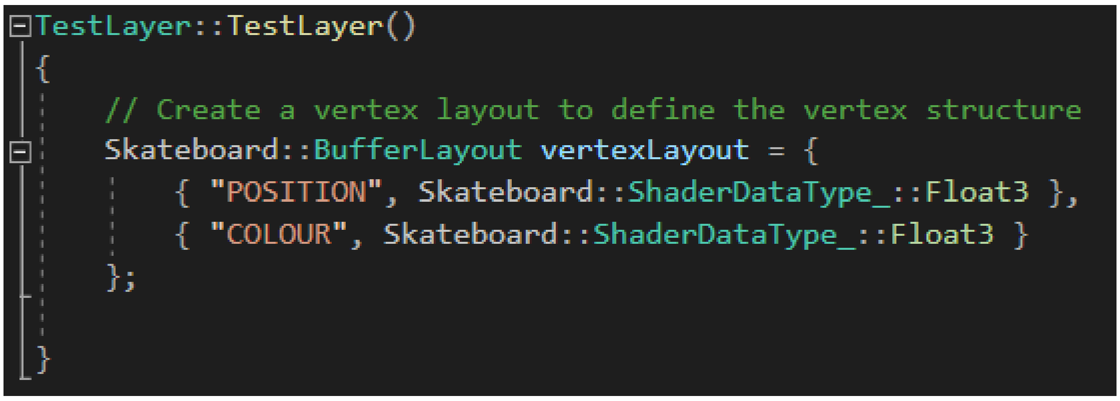
```cpp
#include <Skateboard.h>
#include "Skateboard/EntryPoint.h"
#include "TestLayer.h"

class GameApp : public Skateboard::Application
{
public:
    GameApp()
    {
        // Initialise some layers
        PushLayer(new TestLayer());
    }
};

Skateboard::Application* Skateboard::CreateApplication(int argc, char** argv)
{
    return new GameApp();
}
```

Figure 3 - Pushing the layer into the application stack

Et voilà! We now have correctly set up our application layer and we can start to add some graphics. Go ahead and try to build, nothing should have changed just yet as we did not add anything to our layer. If you encounter issues in the process feel free to ask a tutor and we'll try to get it sorted!

Creating a Vertex & Index Buffer

As you have learned from previous modules, the most common type of rendering is done with rasterization since it remains one of the most efficient methods to draw pixels while utilising the parallelisation of your GPU at its maximum. The Skateboard engine therefore uses these concepts as well to create quick or complex graphics of your choosing.
Before drawing any graphics, we need to create a buffer of vertices and a buffer of indices that will be permanently located on your GPU VRAM for rendering. We don't want to have them on the system RAM as that will require slow data transfers from the CPU to the GPU on your different draw calls. Recall that a vertex buffer is a continuous region of memory containing the data for your vertices (position, normal, etc) and that an index buffer is another region of continuous memory that contains 32 bits integers used by your GPU to select the correct vertices from your buffer.
To help you create the vertex buffer, the Skateboard engine uses a **Layout**, a structure that defines how the data inside your vertex buffer should be interpreted using one or multiple elements. For this simple example, we want to have a vertex layout that contains a position and a colour. The position element will describe where the vertex lies in the virtual environment, and the colour element will describe its colour. Go ahead and create a simple layout in the constructor of your layer (see *Figure 4*). Note: At the time of writing this document, it will be very important that you use the correct semantics for your elements. Refer to *Table 1* when selecting the appropriate semantic for your element.



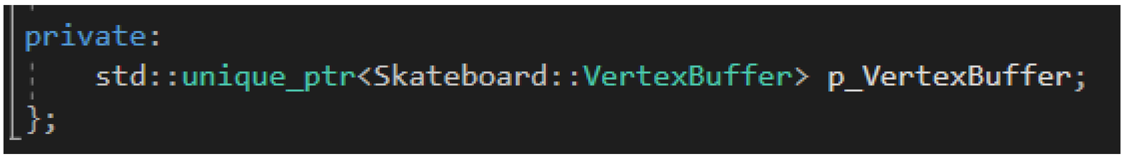
```
TestLayer::TestLayer()
{
    // Create a vertex layout to define the vertex structure
    Skateboard::BufferLayout vertexLayout = {
        { "POSITION", Skateboard::ShaderDataType_::Float3 },
        { "COLOUR", Skateboard::ShaderDataType_::Float3 }
    };

}
```

Figure 4 - Our implementation of the vertex layout with a position and colour element

Element Type	Semantic Name
Position	**POSITION**
Normal	**NORMAL**
Texture Coordinate	**TEXCOORD**
Tangent	**TANGENT**
Bitangent	**BITANGENT**
Colour	**COLOUR** or **COLOR**

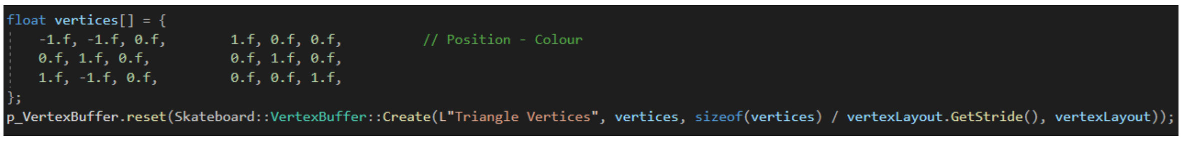
Table 1 - Vertex Layout semantics for each possible element

Now let’s define our triangle vertices with our layout. Recall that the layout we have chosen defines a position followed by a colour for each vertex. Therefore, the data we input will be interpreted in that order. We are going to use a generic C-style array to declare our vertices for simplicity (see *Figure 6*). As mentioned above, these vertices need to be sent to the GPU VRAM. Thankfully for us, the Skateboard engine provides a *Skateboard::VertexBuffer* class that will handle that automatically. In the header of your layer, add a member pointer to this class so it can be used and referenced in all the member functions. Note: in our example we will be using smart pointers to automatically handle the object deletion on exit (see *Figure 5*). We can then create the vertex buffer in the constructor as well by using the *Skateboard::VertexBuffer::Create()* function and simply supplying the vertex data we defined as well as the layout (see *Figure 6*). In the *Skateboard* engine, it is very important to use the *Create()* methods when constructing engine objects as their implementation will vary depending on the platforms. Have a look inside (select the *Create* function and press F12), you will see that on Windows it will instantiate a different VertexBuffer class than Playstation 5, etc. This is also why the constructor of these base engine classes are under the ‘protected’ field, such that only these derived implementations can be instantiated, for example in a *Create()* function.



```
private:
    std::unique_ptr<Skateboard::VertexBuffer> p_VertexBuffer;
};
```

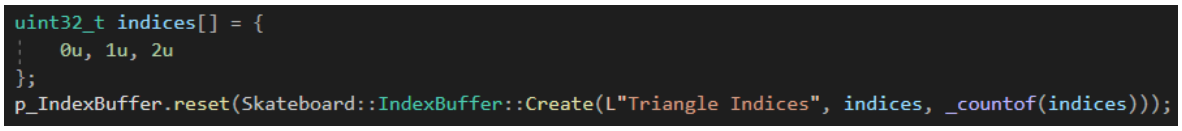
Figure 5 - The Skateboard VertexBuffer member pointer in our TestLayer



```
float vertices[] = {
    -1.f, -1.f, 0.f,        1.f, 0.f, 0.f,          // Position - Colour
    0.f, 1.f, 0.f,          0.f, 1.f, 0.f,
    1.f, -1.f, 0.f,         0.f, 0.f, 1.f,
};
p_VertexBuffer.reset(Skateboard::VertexBuffer::Create(L"Triangle Vertices", vertices, sizeof(vertices) / vertexLayout.GetStride(), vertexLayout));
```

Figure 6 - Vertex declaration and buffer creation

The index buffer will be created using a similar idea, except that we will need to use the *Skateboard::IndexBuffer* class (see *Figure 7*). Remember that the order of vertices for drawing needs to be counterclockwise. Since our vertex buffer declaration already accounts for this, we can just use the continuous indices 0, 1 & 2.



```
uint32_t indices[] = {
    0u, 1u, 2u
};
p_IndexBuffer.reset(Skateboard::IndexBuffer::Create(L"Triangle Indices", indices, _countof(indices)));
```

Figure 7 - Indices declaration and buffer creation

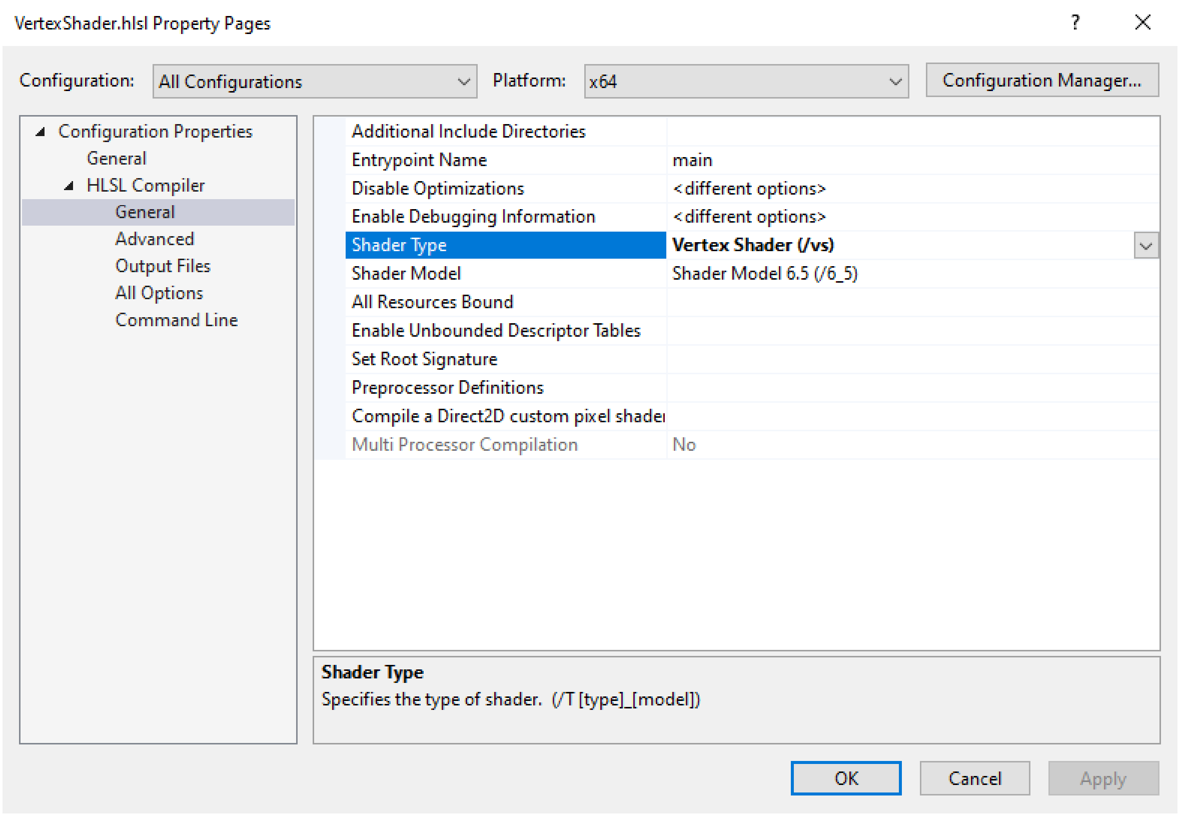
Awesome! Quickly check that everything compiles and we’ll be ready to tackle the next step.

Creating a Rasterization Pipeline & Drawing our Triangle

Working with pipelines is the core concept of graphics programming. As you have seen in the lectures, the traditional rasterization pipeline consists of several stages that will, vulgarly speaking, transform our vertices to pixels on the screen. For our current purposes, we will limit ourselves to the usage of the vertex shader and the pixel shader stages. Note that in order to run properly, the Skateboard engine expects a minimum of a vertex shader stage in the pipeline.

Vertex Shader

Let's start by creating a basic vertex shader! We will first need to add a HLSL file to our project source (right click on *src* > *Add* > *New Item…* > *Installed* > *Visual C++* > *HLSL* > *Vertex Shader File* OR just type *vertexshader.hlsl* in the 'Name' field). We will need to configure our shader by right-clicking on the newly created HLSL file > Properties > Configuration Properties > HLSL Compiler > General. Under *Shader Type* make sure to select *Vertex Shader (/vs)* and under *Shader Model* it should be set to *Shader Model 6.5 (/6_5)*. See *Figure 8*.



VertexShader.hlsl Property Pages

Configuration: All Configurations
Platform: x64

- Configuration Properties
  - General
  - HLSL Compiler
    - General
    - Advanced
    - Output Files
    - All Options
    - Command Line

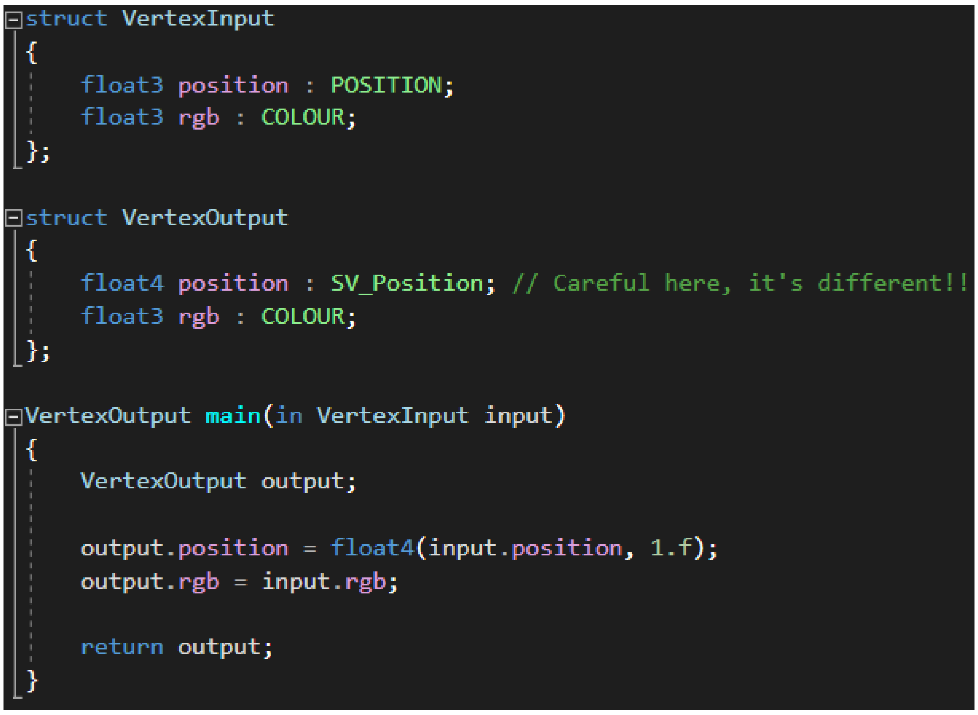
Additional Include Directories	
Entrypoint Name	main
Disable Optimizations	<different options>
Enable Debugging Information	<different options>
Shader Type	**Vertex Shader (/vs)**
Shader Model	Shader Model 6.5 (/6_5)
All Resources Bound	
Enable Unbounded Descriptor Tables	
Set Root Signature	
Preprocessor Definitions	
Compile a Direct2D custom pixel shader	
Multi Processor Compilation	No

Shader Type
Specifies the type of shader. (/T [type]_[model])

Figure 8 - Configuration Properties of the Vertex Shader

In the shader we will need to define a structure for our vertex layout. This will be the input of our main function, as DirectX will provide the data based on the layout we created earlier. This is where the semantic names are important: we need to reference each field of our

structure to the semantic of our layout. DirectX will bind the data to the variables where you add your semantics in the shader file.
We also need to define what goes out of the shader stage, which will be the exact same data as our input for now. However, note how the position data is transformed into a *float4* with the *SV_POSITION* semantic, which is the required format and semantic to send this data to be rasterized (the *COLOUR* data will only be interpolated). See *Figure 9* for the very basic vertex shader we implemented. We will expand on this stage later on when adding the camera.

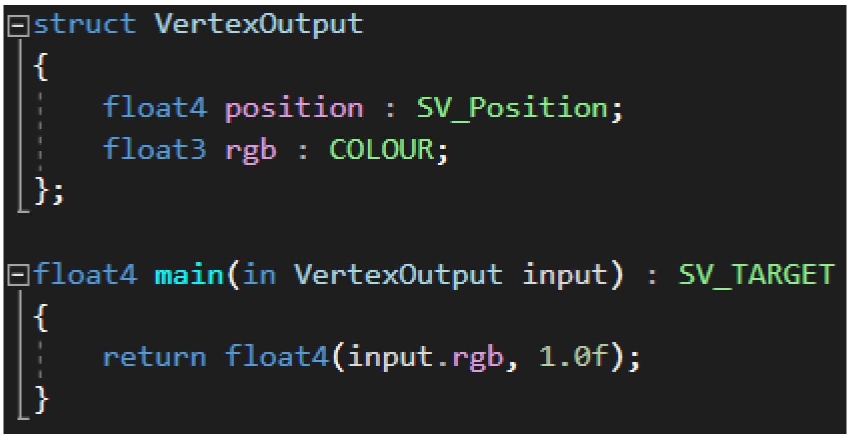


```
struct VertexInput
{
    float3 position : POSITION;
    float3 rgb : COLOUR;
};

struct VertexOutput
{
    float4 position : SV_Position; // Careful here, it's different!!
    float3 rgb : COLOUR;
};

VertexOutput main(in VertexInput input)
{
    VertexOutput output;

    output.position = float4(input.position, 1.f);
    output.rgb = input.rgb;

    return output;
}
```

Figure 9 - Basic vertex shader

Pixel Shader

Just like the vertex shader, create another empty HLSL file and adjust its HLSL Compiler properties to the appropriate *Shader Type* and *Shader Model*. The input data for our pixel shader is quite simply the output data of the vertex shader. So we can copy our VertexOutput struct and paste it in the pixel shader. To avoid copying data around, you could make use of HLSL headers (with extension *.hlsli*) that work in the exact same manner as C++ headers; you can include them using, for instance, *#include “Buffers.hlsli”*. This will be a good idea when you start to have a project with lots of pipelines and shader files that reuse the same data layouts. The pixel shader main function will take this structure as its parameter and will need to return a *float4* to the *SV_TARGET* register to output an RGBA colour on the render target. See *Figure 10* for our basic pixel shader.

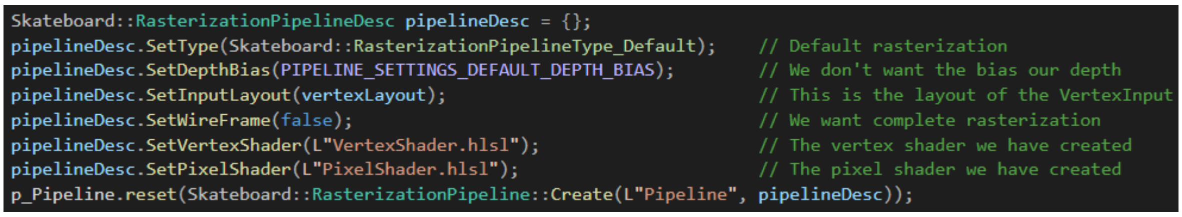


```
struct VertexOutput
{
    float4 position : SV_Position;
    float3 rgb : COLOUR;
};

float4 main(in VertexOutput input) : SV_TARGET
{
    return float4(input.rgb, 1.0f);
}
```

Figure 10 - Basic pixel shader

Creating the Pipeline

Now that our shaders are implemented, we need to create a pipeline in the *Skateboard* engine that will make use of these shaders. Let's head back to our *TestLayer.h* (or your equivalent) and add a pointer to a *Skateboard::RasterizationPipeline*. With that, we can now start to fill in a Skateboard::RasterizationPipelineDesc structure in the constructor. This structure is there to help you define what shaders and resources this pipeline will use. Additionally, you can find some extra settings that may become useful for certain scenarios such as making the pipeline wireframe. *Figure 11* shows how we described a basic rasterization pipeline that will not be wireframed, and use the shaders we just created. Note that failure to provide this minimal amount of information to the description (except for the pixel shader that remain optional) is likely to result in an exception thrown on the pipeline creation. In these events, it is very useful to look at what the API outputs in the Output Window of Visual Studio!

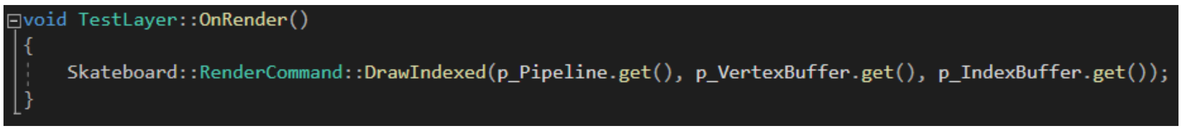


```
Skateboard::RasterizationPipelineDesc pipelineDesc = {};
pipelineDesc.SetType(Skateboard::RasterizationPipelineType_Default);    // Default rasterization
pipelineDesc.SetDepthBias(PIPELINE_SETTINGS_DEFAULT_DEPTH_BIAS);        // We don't want the bias our depth
pipelineDesc.SetInputLayout(vertexLayout);                              // This is the layout of the VertexInput
pipelineDesc.SetWireFrame(false);                                       // We want complete rasterization
pipelineDesc.SetVertexShader(L"VertexShader.hlsl");                     // The vertex shader we have created
pipelineDesc.SetPixelShader(L"PixelShader.hlsl");                       // The pixel shader we have created
p_Pipeline.reset(Skateboard::RasterizationPipeline::Create(L"Pipeline", pipelineDesc));
```

Figure 11 - Filling up the pipeline description and creating the pipeline

Drawing the Triangle

The very last step is now to tell our engine to draw this triangle using this pipeline inside the *Render()* function of our layer. This is done by using a *Skateboard* render command. These commands are your tools to emit different kinds of draw calls depending on your goals. Here we will use the *Skateboard::RenderCommand::DrawIndexed()* interface as shown on *Figure 12*. It draws our mesh based on the indices we have created earlier.



```
void TestLayer::OnRender()
{
    Skateboard::RenderCommand::DrawIndexed(p_Pipeline.get(), p_VertexBuffer.get(), p_IndexBuffer.get());
}
```

Figure 12 - Drawing the triangle

Now compile and run the project. If everything went well, congrats! You now have done your first steps into graphics programming and obtained a working rendering of your triangle!

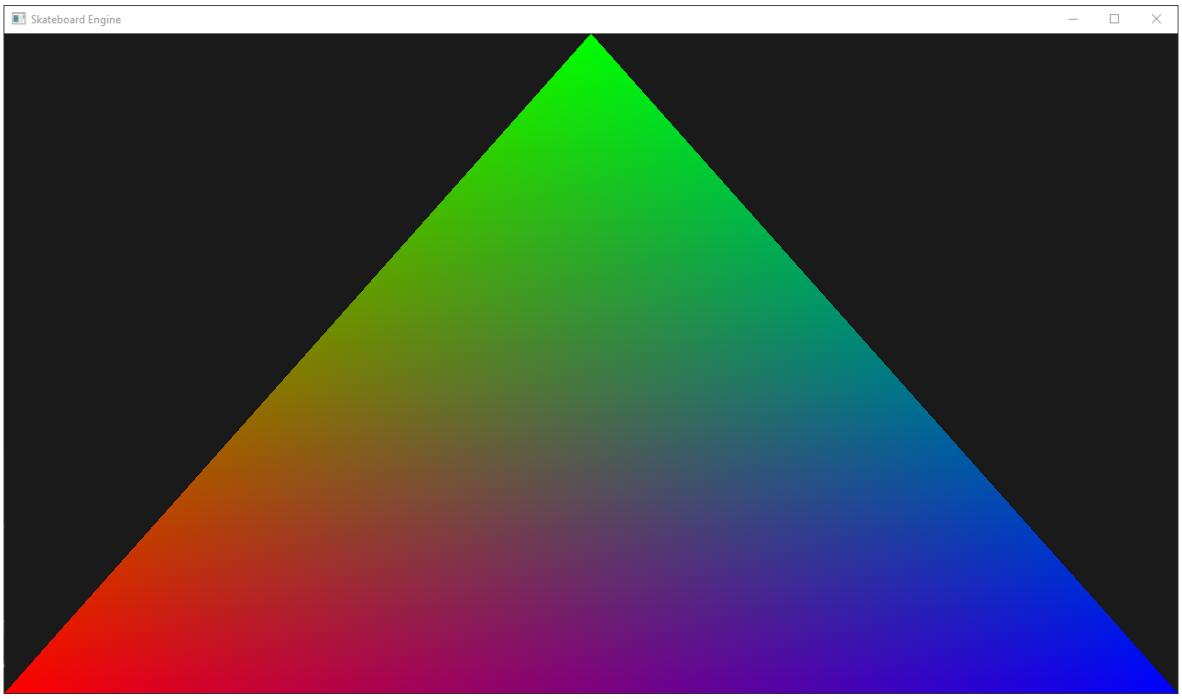


Figure 13 - Resulting triangle

Adding a Perspective Camera

Let's take things a little bit further and add a controllable perspective camera in our virtual environment. This will teach you how to transfer custom data from the CPU to the GPU. In fact, we will need to retrieve and send the view and projection matrix of our camera to the vertex shader stage in order to transform our triangle vertices accordingly.

Creating the camera

Let's start by adding a *Skateboard::PerspectiveCamera* in our layer's header file and initialise it in the CPP with some default settings. You can either use the constructor's initialiser list of your layer or use the *PerspectiveCamera::Build()* member function. We used the initialiser list to create a camera with settings displayed in *Table 2*. Note that the aspect ratio is obtained using the graphics context, a singleton instance that holds information related to the output screen and graphics.

Fov	0.25 * SKTBD_PI (45 degrees)
Aspect Ratio	Skateboard::GraphicsContext::Context->GetClientAspectRatio()
Near Plane	0.1
Far Plane	1000
Initial Position	0, 0, -10
Initial Target	0, 0, -9
Initial Up	0, 1, 0

Table 2 - Default camera settings

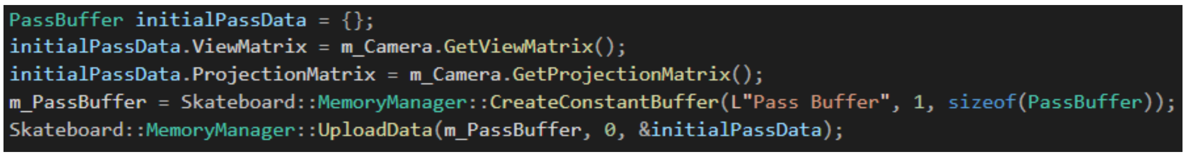
Creating a Constant Buffer on the Pipeline

In order to transfer the data from the CPU to the GPU, we will need to use upload buffers. As their name describes, their purpose is to upload data to the GPU VRAM so it can be read. Recall that this operation is extremely slow and is likely to be the performance bottleneck of your graphics application, so **make sure to use upload buffers for data that is likely to change between frames**. Otherwise, for data that will never change, you would prefer using a default buffer. Another challenge with upload buffers is that we cannot change its data while it is being used by the GPU. This is a concept of its own referred to as *frame resources* for your own reading. Luckily for us, the engine should already account for this with the help of the Skateboard::*MemoryManager*. You are free to create upload buffers without it, but this manager will ensure that you will not modify the data that is in use. When using this manager, we will track our upload buffers using unsigned integer IDs. So let's go ahead and add a member *uint32_t m_PassBuffer* to our layer header in the private field. We call it *PassBuffer* as it will hold data that is valid for all geometries (we will almost always consider our camera when rendering). We will also need to define a custom structure for our

PassBuffer in the header. Our following implementation defines a very simple buffer that only considers the camera view and projection matrices:

```
struct PassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
};
```

However, in the future you might find it useful to send many more pieces of information such as deltatime, inverse matrices, etc.
As mentioned above, we will then need to create our buffer using the memory manager. Use the *Skateboard::MemoryManager::CreateConstantBuffer()* method to create a constant buffer and Skateboard the *Skateboard::MemoryManager::UploadData()* method to upload some initial data. See *Figure 14* as our example for sending the initial camera data after creating the buffer.



```
PassBuffer initialPassData = {};
initialPassData.ViewMatrix = m_Camera.GetViewMatrix();
initialPassData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
m_PassBuffer = Skateboard::MemoryManager::CreateConstantBuffer(L"Pass Buffer", 1, sizeof(PassBuffer));
Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &initialPassData);
```

Figure 14 - Initialisation of the camera

Now that this buffer has been created with some initial data, we will need to tell our pipeline to bind it to the vertex shader. This is done using the *AddConstantBufferView()* method on the pipeline description. For this example, we will bind it to the shader register 0 and space 0. These are not very important to understand at this time, but you can think of the shader registers and spaces as means to identify where to look for some data you are sending. See the code snippet below for our addition of the constant buffer to the pipeline:

```
pipelineDesc.AddConstantBufferView(
	Skateboard::MemoryManager::GetUploadBuffer(m_PassBuffer),
	0, // register
	0, // space
	Skateboard::ShaderVisibility_VertexShader // restricted to the vertex shader
);
```

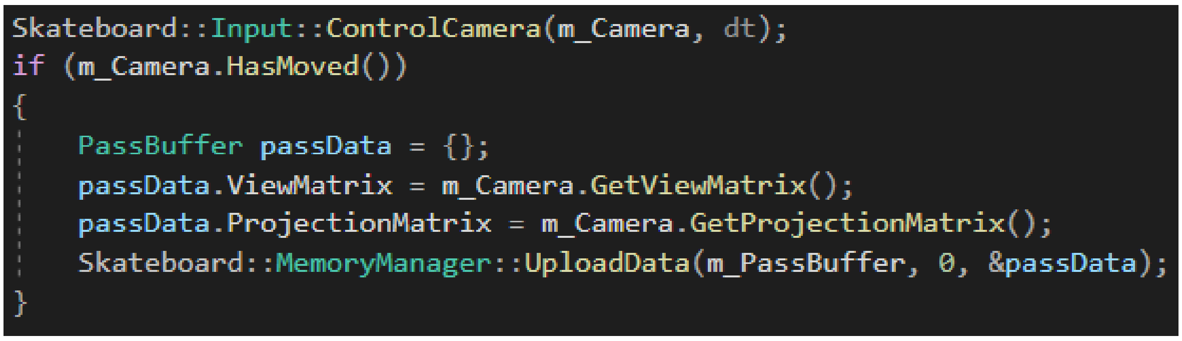
Handling Inputs and Updating

The next step we need to consider is controlling the camera and updating the data in our buffer when the camera has moved. Let's head over to the layer's *HandleInputs()* function and add:

```
Skateboard::Input::ControlCamera(m_Camera, dt);
```

While this remains valid at the time of writing this document, it may be updated and handled differently in future versions of the engine. Essentially, our input system defines a custom implementation to move our camera. This however isn't ideal as you may wish to control the

camera differently, or have multiple camera controllers. In the future, such controllers may have been implemented and therefore you should be careful to either use this new way of controlling your camera or otherwise create your own.
Once our camera has been updated, we can detect whether it has moved or not using the *PerspectiveCamera::HasMoved()* method. If so, we need to upload the new view matrix to our constant buffer for the change to take effect in our shader. This is done in the exact same way we have initialised our buffer, see *Figure 15*.



```
Skateboard::Input::ControlCamera(m_Camera, dt);
if (m_Camera.HasMoved())
{
    PassBuffer passData = {};
    passData.ViewMatrix = m_Camera.GetViewMatrix();
    passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
    Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &passData);
}
```

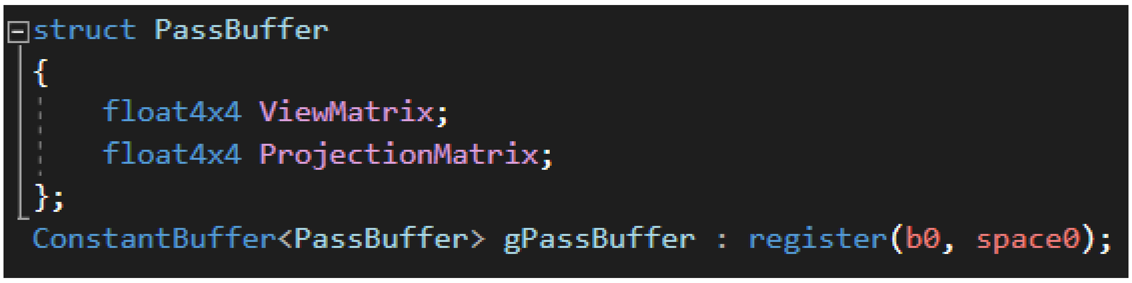
Figure 15 - Uploading the updated data of our camera upon movement

We also need to consider that the window may be resized by the user, either by dragging the edges of the window or by making it fullscreen. These actions will cause the aspect ratio to change, and therefore our camera projection matrix needs to update accordingly. Thankfully for use, the *Layer::OnResize()* function is called when such resizing occurs on the window, and we can therefore add the following snippet inside our *OnResize()* function to immediately account for the change:

```
m_Camera.OnResize(newClientWidth, newClientHeight);
PassBuffer passData = {};
passData.ViewMatrix = m_Camera.GetViewMatrix();
passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &passData);
```

Updating the Vertex Shader

The final step is to now consider this data in our vertex shader. In other words, even though we have set the code up to send the camera data to the GPU, we have yet to tell it to use it. Let's head back to our vertex shader HLSL file and add a struct for our *PassBuffer* with the exact same data as declared on the C++ side. Because of how the *Skateboard* engine defines data types, this should be a matter of a simple copy/paste. Note that you could also include C++ headers in your HLSL files and have your buffers available on both ends at once. We then need to tell HLSL to locate this constant buffer in the register 0 and space 0, as it is where we have been sending it from the pipeline description. This is done using the HLSL *ConstantBuffer* keyword as demonstrated in *Figure 16*.



```
struct PassBuffer
{
    float4x4 ViewMatrix;
    float4x4 ProjectionMatrix;
};
ConstantBuffer<PassBuffer> gPassBuffer : register(b0, space0);
```

Figure 16 - Adding the constant buffer to the vertex shader

We can then use this information in our vertex shader *main()* function by accessing our *gPassBuffer* variable. Replace your output position line with the following snippet:

```
output.position = mul(gPassBuffer.ViewMatrix, float4(input.position, 1.f));
output.position = mul(gPassBuffer.ProjectionMatrix, output.position);
```

This will transform the vertices of our triangle according to our camera view and projection. Essentially, the view matrix will shift the world's origin and direction to our camera view, and the projection matrix will apply a projection transformation such that depth is perceived in a similar way to the real world. With these adjustments, compile and run the project. You should now be able to control your camera with W,A,S,D and change your view by right-clicking on your mouse and dragging it!

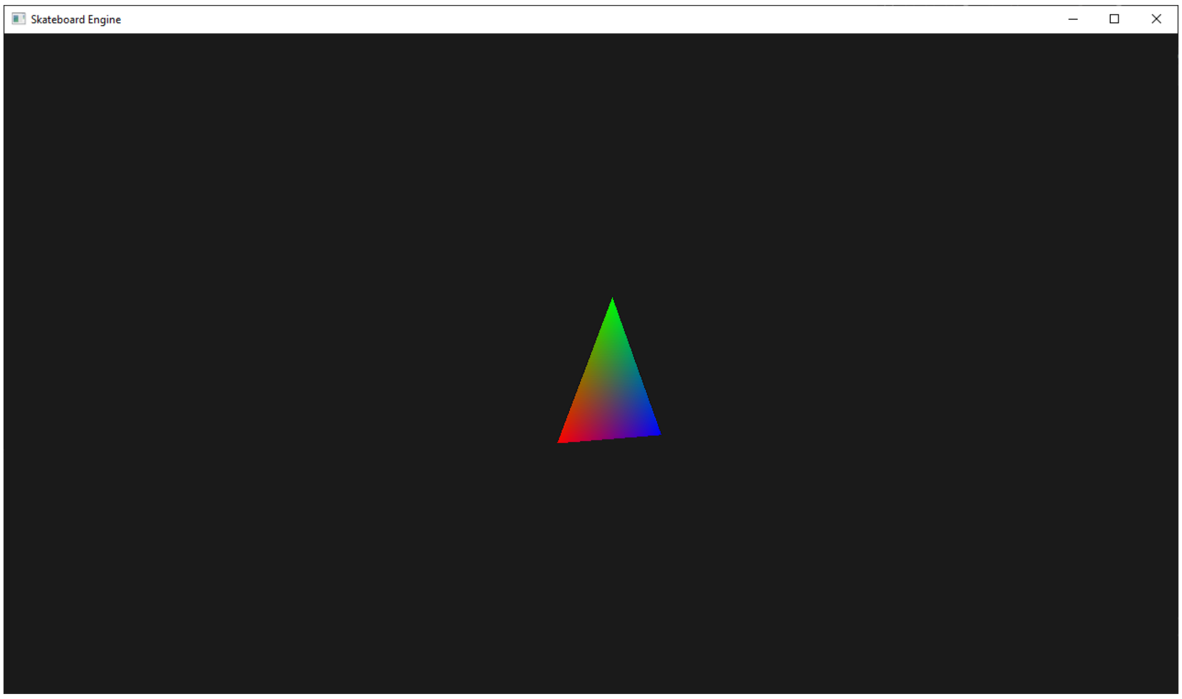


Figure 17 - Transformed triangle after moving the camera and view

My First User Interface

Having a user interface for real-time debugging is very useful in game development, so much so that such user interfaces have been observed to be used in AAA game engines such as *Grand Theft Auto 6*'s leaked footage from September 18, 2022. The goal is to ease your testing by using buttons and sliders to change values in real-time instead of having to recompile your application every time. For this exercise, we will write a little user interface using the *ImGui* libraries to modify the position and rotation of our camera with sliders.
Let's get started! The first thing we need to identify is the *Layer::OnImGuiRender()* function. This is the space where you will write your user interface. Essentially, this function gets called for all layers after the main rendering has occurred, such that the user interface always remains visible on top of the view of our virtual world. Inside this function, add the following snippet:

```
ImGui::Begin("Options");
ImGui::Text("Camera Options:");
ImGui::End();
```

Evidently, *ImGui::Begin(<window title>)* starts a new window with the title you have inputted. You can think of it as the equivalent of entering in a scope in C++ with '*{*'. Every *Begin()* must be terminated with an *End()*, otherwise *ImGui* will not know when to terminate this scope. On Windows, we are able to drag this window around and even dock it to our main window. However, on Playstation we do not necessarily want to do that because we would be trying to control our UI with a gamepad. On top of that, *ImGui* windows may be created on top of each other if we do not specify initial positions and sizes. While that is fine on the Windows platform, as this information will be internally stored after a modification, this may become more challenging on Playstation. For these reasons, it will be good practice to specify a position and size to be considered on the first frame, and *ImGui* allows us to do this with a flag we need to input in our functions. We will use *ImGui::SetNextWindowSize()* and *ImGui::SetNextWindowPos()* with the *ImGuiCond_FirstUseEver* on top of our *Begin()* function to initialise our window on the very first time they are being used:

```
ImGui::SetNextWindowSize(ImVec2(400, 200), ImGuiCond_FirstUseEver);
ImGui::SetNextWindowPos(ImVec2(20, 20), ImGuiCond_FirstUseEver);
```

If you already compiled and ran the initial, empty window, then this change will have no effect as *ImGui* will already have registered and saved this window internally and will not consider it as a 'first use ever'. In that case, you may test with the *ImGuiCond_Appearing* flag.
Adding our camera controls will be very easy. All we have to do is to use the *ImGui::DragFloat3()* function with our data to be modified in it. You may notice that this function returns *true* when data has been modified from our sliders. When that's the case, we will want to update our camera and send the new data to our constant buffer.

```
float3 camPos = m_Camera.GetPosition();
float3 camRot = m_Camera.GetRotation();

bool cameraModified = false;
```

```
cameraModified |= ImGui::DragFloat3("Camera Position", &camPos.x, .1f);
cameraModified |= ImGui::DragFloat3("Camera Rotation", &camRot.x, .1f);
if (cameraModified)
{
	m_Camera.SetPosition(camPos);
	m_Camera.SetRotation(camRot);
	m_Camera.UpdateViewMatrix();
	PassBuffer passData = {};
	passData.ViewMatrix = m_Camera.GetViewMatrix();
	passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
	Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &passData);
}
```

Note that we will require to manually call *PerspectiveCamera::UpdateViewMatrix()* to regenerate the view matrix after the change has been sent. With that, you have made your first steps into writing debugging user interface! *ImGui* is a very powerful and complete library that offers much more than what we have covered, feel free to add and check out the *ImGui::ShowDemoWindow()* in your code to see all the potential.

(Bonus) Switching to Playstation 5

Since the ***GameApp* project** is supposed to be abstracted for all platforms, most of our work is already done and we can just switch the solution configuration from **Windows** to **Prospero**. However, once you've done that and tried to run it, you might notice that there was a little catch! The Playstation 5 cannot understand and compile HLSL files! We will need to write an additional set of shaders for their PSSL (PlayStation Shading Language) equivalents. For our purposes, they are essentially an exact copy of the HLSL code counterparts, except for the removal of the semantics from the VertexInput structure and for an extra addition: we now need to define a Shader Resource Table (SRT) Signature. This signature will tell the engine how to send the data properly to the shader based on the shader register and register space. This concept is explained in more detailed in the Playstation devnet, but for now just consider this extra signature to be added inside your PSSL vertex shader file:

```
SrtSignature gSignature
{
        CBV(b0, space=0)
}
```

In order to tell our shader to use this signature, we will need to add a custom identification on top of the main function:

```
[SrtSignature(gSignature)]
VertexOutput main(uint id : S_VERTEX_ID) { … }
```

Notice how the main function now doesn't consider the input vertex as a parameter, but rather only the vertex ID supplied from the index buffer. The Playstation 5 pipeline does not necessarily assemble the vertex buffer with an index buffer like DirectX or the Playstation 4. It is considered a better practice to supply it as a *RegularBuffer* instead and use the index ourselves to manually retrieve the vertex. Essentially, it ends up being the exact same thing but just leaves us one extra step to consider. The engine will supply this buffer for you automatically and therefore does not need to be part of the SRT Signature. See the code snippet below and *Appendix E* for the full and final shader:

```
…
RegularBuffer<InputVertex> gVertices;
…

Output main(uint id : S_VERTEX_ID)
{
        VertexInput input = gVertices[id];
        …
}
```

Similar modifications need to be made to the *PixelShader.pssl* file after porting the HLSL code. See *Appendix F* for the full implementation. You should add a PSSL file in the exact same way you would add a HLSL file to your project, and after that you should check that it will be compiled as the appropriate shader type in its PSSL Compiler properties.

Appendix A - TestLayer.h

```
#pragma once
#include <Skateboard.h>

struct PassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
};

// The 'final' keyword tells the compiler that this is the last derivation of this
// class, and can potentially consider more optimisations in Release & Ship modes
class TestLayer final : public Skateboard::Layer
{
public:
	TestLayer();
	~TestLayer() final override;

	virtual void OnResize(int newClientWidth, int newClientHeight) final override;		//
Called when the platform called for a resize of the main application
	virtual void OnAttach() final override;
					// Called when this layer is being attached to the main
application (PushLayer)
	virtual void OnDetach() final override;
						// Called when this layer is being detached from the
main application (PopLayer)
	virtual bool OnHandleInput(float dt) final override;
		// Used to handle the inputs according to your game logic
	virtual bool OnUpdate(float dt) final override;
			// Used to update your scene and objects
	virtual void OnRender() final override;
						// Used to render your scene and objects
	virtual void OnImGuiRender() final override;
			// Used to render your user interface

private:
	std::unique_ptr<Skateboard::VertexBuffer> p_VertexBuffer;
	std::unique_ptr<Skateboard::IndexBuffer> p_IndexBuffer;
	std::unique_ptr<Skateboard::RasterizationPipeline> p_Pipeline;
	Skateboard::PerspectiveCamera m_Camera;
	uint32_t m_PassBuffer;
};
```

Appendix B - TestLayer.cpp

```
#include "TestLayer.h"

TestLayer::TestLayer() :
	m_Camera(.25f * SKTBD_PI,
Skateboard::GraphicsContext::Context->GetClientAspectRatio(), .1f, 1000.f, float3(0.f, 0.f,
-10.f), float3(0.f, 0.f, -9.f), float3(0.f, 1.f, 0.f))
{
	// Create a vertex layout to define the vertex structure
	Skateboard::BufferLayout vertexLayout = {
		{ "POSITION", Skateboard::ShaderDataType_::Float3 },
		{ "COLOUR", Skateboard::ShaderDataType_::Float3 }
	};

	float vertices[] = {
		-1.f, -1.f, 0.f,		1.f, 0.f, 0.f,			// Position - Colour
		0.f, 1.f, 0.f,			0.f, 1.f, 0.f,
		1.f, -1.f, 0.f,			0.f, 0.f, 1.f,
	};
	p_VertexBuffer.reset(Skateboard::VertexBuffer::Create(L"Triangle Vertices",
vertices, sizeof(vertices) / vertexLayout.GetStride(), vertexLayout));

	uint32_t indices[] = {
		0u, 1u, 2u
	};
	p_IndexBuffer.reset(Skateboard::IndexBuffer::Create(L"Triangle Indices", indices,
_countof(indices)));

	PassBuffer initialPassData = {};
	initialPassData.ViewMatrix = m_Camera.GetViewMatrix();
	initialPassData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
	m_PassBuffer = Skateboard::MemoryManager::CreateConstantBuffer(L"Pass
Buffer", 1, sizeof(PassBuffer));
	Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &initialPassData);

	Skateboard::RasterizationPipelineDesc pipelineDesc = {};
	pipelineDesc.SetType(Skateboard::RasterizationPipelineType_Default);	// Default
rasterization
	pipelineDesc.SetDepthBias(PIPELINE_SETTINGS_DEFAULT_DEPTH_BIAS);
	// We don't want the bias our depth
	pipelineDesc.SetInputLayout(vertexLayout);
		// This is the layout of the VertexInput
	pipelineDesc.SetWireFrame(false);
			// We want complete rasterization
	pipelineDesc.SetVertexShader(L"VertexShader.hlsl");
		// The vertex shader we have created
	pipelineDesc.SetPixelShader(L"PixelShader.hlsl");
	// The pixel shader we have created

pipelineDesc.AddConstantBufferView(Skateboard::MemoryManager::GetUploadBuffer(m_
PassBuffer), 0, 0, Skateboard::ShaderVisibility_VertexShader);
```

```
	p_Pipeline.reset(Skateboard::RasterizationPipeline::Create(L"Pipeline",
pipelineDesc));
}

TestLayer::~TestLayer()
{
}

void TestLayer::OnResize(int newClientWidth, int newClientHeight)
{
	m_Camera.OnResize(newClientWidth, newClientHeight);
	PassBuffer passData = {};
	passData.ViewMatrix = m_Camera.GetViewMatrix();
	passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
	Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &passData);
}

void TestLayer::OnAttach()
{

}

void TestLayer::OnDetach()
{

}

bool TestLayer::OnHandleInput(float dt)
{
	Skateboard::Input::ControlCamera(m_Camera, dt);
	if (m_Camera.HasMoved())
	{
		PassBuffer passData = {};
		passData.ViewMatrix = m_Camera.GetViewMatrix();
		passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
		Skateboard::MemoryManager::UploadData(m_PassBuffer, 0, &passData);
	}

	return true;
}

bool TestLayer::OnUpdate(float dt)
{
	return true;
}

void TestLayer::OnRender()
{
	Skateboard::RenderCommand::DrawIndexed(p_Pipeline.get(),
p_VertexBuffer.get(), p_IndexBuffer.get());
}

void TestLayer::OnImGuiRender()
{
	ImGui::SetNextWindowSize(ImVec2(400, 200), ImGuiCond_FirstUseEver);
	ImGui::SetNextWindowPos(ImVec2(20, 20), ImGuiCond_FirstUseEver);
	ImGui::Begin("Options");
	{
		ImGui::Text("Camera Options:");
		float3 camPos = m_Camera.GetPosition();
		float3 camRot = m_Camera.GetRotation();

		bool cameraModified = false;
		cameraModified |= ImGui::DragFloat3("Camera Position", &camPos.x, .1f);
		cameraModified |= ImGui::DragFloat3("Camera Rotation", &camRot.x, .1f);
		if (cameraModified)
		{
			m_Camera.SetPosition(camPos);
			m_Camera.SetRotation(camRot);
			m_Camera.UpdateViewMatrix();
			PassBuffer passData = {};
			passData.ViewMatrix = m_Camera.GetViewMatrix();
			passData.ProjectionMatrix = m_Camera.GetProjectionMatrix();
			Skateboard::MemoryManager::UploadData(m_PassBuffer, 0,
&passData);
		}
	}
	ImGui::End();

}
```

Appendix C - VertexShader.hlsl

```
struct VertexInput
{
	float3 position : POSITION;
	float3 rgb : COLOUR;
};

struct VertexOutput
{
	float4 position : SV_Position; // Careful here, it's different!!
	float3 rgb : COLOUR;
};

struct PassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
};
ConstantBuffer<PassBuffer> gPassBuffer : register(b0, space0);

VertexOutput main(in VertexInput input)
{
	VertexOutput output;
	output.position = mul(gPassBuffer.ViewMatrix, float4(input.position, 1.f));
	output.position = mul(gPassBuffer.ProjectionMatrix, output.position);
	output.rgb = input.rgb;

	return output;
}
```

Appendix D - PixelShader.hlsl

```
struct VertexOutput
{
		float4 position : SV_Position;
		float3 rgb : COLOUR;
};

float4 main(in VertexOutput input) : SV_TARGET
{
		return float4(input.rgb, 1.0f);
}
```

Appendix E - VertexShader.pssl

```
struct VertexInput
{
	float3 position;
	float3 rgb;
};

struct VertexOutput
{
	float4 position  : S_POSITION;
	float3 rgb : COLOUR;
};

struct PassBuffer
{
	float4x4 ViewMatrix;
	float4x4 ProjectionMatrix;
};
ConstantBuffer<PassBuffer> gPassBuffer : register(b0, space0);
RegularBuffer<InputVertex> gVertices;

SrtSignature gSignature
{
	CBV(b0, space=0)
}

[SrtSignature(gSignature)]
VertexOutput main(uint id : S_VERTEX_ID)
{
	VertexOutput output;
	InputVertex input = gVertices[id];

	output.position = mul(gPassBuffer.ViewMatrix, float4(input.position, 1.f));
	output.position = mul(gPassBuffer.ProjectionMatrix, output.position);
	output.rgb = input.rgb;

	return output;
}
```

Appendix F - PixelShader.pssl

```
struct VertexOutput
{
	float4 position : S_POSITION;
	float3 rgb : COLOUR;
};

SrtSignature gSignature
{
	// Empty for now as we are not sending any data to the pixel shader!
}

[SrtSignature(gSignature)]
float4 main(in VertexOutput input) : S_TARGET_OUTPUT0
{
	return float4(input.rgb, 1.0f);
}
```