**Instructions**Provide the code and files that were modified and video recording when working with render doc for each part that is applicable in separate folders named accordingly (Part 1, Part2 etc.). Each separate part should work and provide the same image as shown at the end of each part below. Images below sections (1b, 1d etc.) also should match when that section is done. Add comments to the new code to which section of part it belongs if not specified already.

### section A

### Task A1

Get started by just setting up your project. Just copy one of the previous works starting files as the base code.

I recommend you delete all the TODO: comments from the base code so it doesn’t confuse anything later.

Make sure you edit and re-run the CMakeLists.txt file to accurately name your project solution. Edit the window title bar and background color to be something different as well:

A green triangle on a green background

Description automatically generated

### task A2

Include the Tiny GLTF library in the CMakeList.txt file.

If you look carefully in CMakeLists.txt, you will see how it searches the folder for certain files.

To make use of the TinyGLTF loading library, extend the project to search its folder.

A black background with white letters

Description automatically generated

After saving the CMakeLists.txt file and recompiling, Visual Studio should prompt you to reload the project. (Do so)

Once the project includes the above files, **#include** the loader and make an instance of the class in Renderer.

**IMPORTANT:** When including TinyGLTF it does expect you to declare some **#defines** before everything in the library will work/link properly (Similar to Gateware.h). Unlike Gateware though, these defines tell the library where to copy the implementation so you should only ever do this inside one **cpp** file. (others can still include it though)

A screen shot of a computer code

Description automatically generated

### task A3

Use the TinyGLTF library to load the rudimentary GLTF from file. (Use it to load the included “triangle.gltf”)

<https://github.com/syoyo/tinygltf?tab=readme-ov-file#loading-gltf-20-model>

The above link shows you how to do this. This should be one of the first things you do during initialization.

To showcase the loader is working, print some information about the 3D model to the console.

***Tip:*** *Try to find & print some of the info you can see by opening the* ***.gltf*** *file in notepad or visual studio.*

### section b

### task b1

The GLTF holds all geometric data in one giant block of memory with formats and offsets encoded in the file. This means the index buffer and vertex buffer are combined.

Traditionally, we would need to process this data and split it into multiple data sets for submission. However, Vulkan supports a very flexible resource system. Which means if Vulkan knows where data is (and how it’s formatted) it can use it.

To create a **unified** vkBuffer, just **OR** | both **bit flags** for VERTEX & INDEX together when creating it. This tells Vulkan you can use the one buffer for both data sets.

I suggest renaming/merging functions like CreateVertex… and CreateIndex… Into **CreateGeometry…**

Once you swap over to using this buffer, things may disappear, but that is to be expected until we adjust rule of 3.

A green rectangular object with white border

Description automatically generated

***Tip:*** *Check how to properly access and copy raw data from an std::vector.*

### task b2

Even though our starting model is very simple, pretty much all GLTF data is going to use an index buffer.

Bind the new unified index buffer & switch to using vkCmdDrawIndexed() to draw all the indices in mesh 0.

Look inside the GLTF file and make a note of what format the index data uses, and where exactly it starts in the unified vertex + index buffer.

A screenshot of a computer screen

Description automatically generated

This information is critical to know/apply when binding the unified buffers to the graphics pipeline.

Never hard-code this information. Always grab it directly from the TinyGLTF Model object.

### task b3

Now we can adjust the **rule of three**, starting with the vertex attributes. Note the **“type”** of the buffer view above.

A commonly missed part of the rule of three is the **vertex stride**. Be sure to account for that change as well.

### task b4

Complete the rule of three adjustment in the **vertex shader**. Then view in RenderDoc:

A screenshot of a computer

Description automatically generated

You may not be able to see it on screen, but the triangle should be working fine in the mesh viewer.

### task b5

GLTF geometry uses a front-face **COUNTER\_CLOCKWISE** winding order. Adjust the **rasterizer stage** so that the incoming triangle is not culled/rejected automatically. This optimization is called back-face culling.

A green and yellow triangle

Description automatically generated

### section c

### task c1

Import the **triangle.gltf** model into the Blender 3D modeling software. We will be using Blender to **homogenize** GLTF 3D models of various data combinations we may come across. This way, we won’t have to write a bunch of branching code. For Example: GLTF allows for a variety of different vertex formats, however that can make writing consistent rendering code a challenge.

If we were writing a tool, then supporting a variety of kinds of vertex formats gives us more flexibility.

### task c2

To achieve this, we have provided the **Custom\_GLTF\_Exporter.py** Blender python script. To use it, you can copy it into the **Script Tab** in Blender and run it once. This will install the exporter; you should see it in the list of available exporters after you run the script. (It is also possible to permanently install the exporter if you wish)

Export the model (repo Models folder) to a different GLTF file (ex: **blender\_triangle.gltf**) using the 3DCC exporter. Open the new GLTF file in a text editor to see how it has changed from the original version.

***Tip:*** *Take some time to review the python script, its main goal is to ensure consistency across different GLTF files.*

### task c3

Once you have your new version of the triangle, switch the loading code over to using it.

When you do this, it may no longer Draw. This is somewhat expected since our 3DCC exporter produces a much more robust vertex format that we will be using from here on out.

A screen shot of a computer code

Description automatically generated

This format contains everything we need for more advanced rendering techniques.

### task c4

As you can probably tell, the **Rule of Three** is no longer for just a 3-float position component.

Edit the rule of three to handle the newly exported 3DCC GLTF vertex format:

Adjust your Vertex and Fragment Shaders to input/output the new attributes.

Edit the **VkVertexInputBindingDescription** so it supports 4 descriptions instead of 1.

Add 3 more **VkVertexInputAttributeDescription**(s) that utilize the 4 new bindings.

You may have noticed tinygltf::BufferView(s) have an **offset** member to where the data is actually stored in the unified buffer. In Vulkan, you don’t specify these offsets during pipeline setup. Instead, go to where you are **binding** the vertex and index buffers before drawing and use it there.

Keep in mind that each vertex attribute in the unified buffer is split-up (aka: non-interleaved data):



This means each attribute starts at a different location in the buffer and must be accurately offset during binding. Once you get this setup correctly, your vertex assembly stage should look like this:

A screenshot of a computer

Description automatically generated

Again. Do not hard code the offsets/strides/lengths of the **Buffers**. Otherwise, no other 3D models will work.

***NOTE:*** *RenderDoc will NOT debug incoming attributes unless they are passed into/used in the shader.*

***Tip:*** *The tinygltf::Accessor class has a built-in* ***ByteStride()*** *function to help you calculate it for each description.*

### task c5

Confirm that your triangle appears properly in RenderDoc (like it did before) and that it draws correctly:

A green and yellow triangle

Description automatically generated

### section d

### task d1

In the last section we will be adding a more detailed 3D model. Once that’s in, we will enhance the application by porting over the camera and lighting code from last week.

In Blender, import 3D model file “Bebe.glb” or generate your own model at: <https://lumalabs.ai/genie?view=create>

Export it from Blender, using our custom 3DCC exporter like we did for the basic triangle.

This will guarantee our new model’s vertex format is consistent with the rendering code we just wrote.

Take note of the exported **Textures** folder. We won’t be using it for this project, but it will become the focus of next project.

### task d2

Confirm the new mesh is loading both visually and in RenderDoc:

A yellow cat on a green background

Description automatically generated

If you didn’t hard-code anything in a prior step, then the geometry should render correctly. Some models may not fully fit inside NDC Space, but you will still be able to preview their geometry in RenderDoc just fine.

A screenshot of a computer

Description automatically generated

### task d3

Carefully port over the **3D matrix &** **uniform buffer** code from **previous project (the one you delivered previously)**.

* VM, PM, Uniform Interfaces, Descriptor Interfaces, SHADER\_VARS in C++ & HLSL, Camera.h

Remove any world matrices, we will just assume all meshes are at identity for now.

You will need to bring the code for allocating the uniform buffers and adjusting **CreatePipelineLayout()**.

Call the camera code and be sure to **update** the uniform buffer each frame:

A yellow silhouette of a chicken

Description automatically generated

Default camera starts at the origin. Move the camera backwards and you will notice the cat is upside down.

### task d4

The starting template code you used is still automatically **flipping the Y** in the shader. Either disable the flip with a Vulkan projection or use a DirectX projection with the Y still flipped. (This is a **shaderc** compiler setting)

After fixing the camera code or adjusting shader compiler settings:

A yellow cat on a green background

Description automatically generated

You should now have full 3D camera control of your GLTF scene!

### task d5

Port the lighting code attached below, this should be *mostly* shader work. You will notice there are no more OBJ\_ATTRIBUTES or other “Classic” material information in the file. Because this is a PBR or “Modern” style of material, all that info is stored in textures.

We will address textures and modern materials in the next project. For now, just hard-code the following **classic material** in the Fragment Shader:

A computer screen with numbers and symbols

Description automatically generated

You will however need to transfer the sun’s lighting info (directional light) and the camera’s position to the SHADER\_VARS struct for this to work. Once you do so, the rest of the code should port without too much issue.

A black cat on a green background

Description automatically generated

Once you get the code ported over and running it may look rather strange…

This is because we are looking at the inside of the 3D model. If you recall, we originally had to change the winding order of the rasterizer pipeline stage to see our initial triangle.

Now that we are working with our custom exporter, we can switch back to clockwise triangles. This technically not correct, but it will work for now until we can incorporate world matrices to account for right-handed coordinates.

A yellow cat on a green background

Description automatically generated

That is much more like it. Don’t mind the odd (slightly melty) geometry, that is just an artifact of the early 3D AI modeling tool used to generate this 3D model. That will improve drastically in the future, I am sure.

As you fly around make sure the Specular highlights shift with the camera. If they do not, that is likely because you are not updating the world space camera position in the shader properly.

Feel free to adjust/play with the lighting variables to make sure everything works as expected.

A silhouette of a cat

Description automatically generated

Just Messing About: Red overhead light reflecting off a shiny Onyx cat statue.