**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.

# Introduction

You will explore techniques required to render GLTF models at a higher fidelity. You will start by loading an official GLTF test model with a more complex PBR material. You will then adapt your drawing code to accurately/correctly render models by accounting for the standard (right-handed) GLTF coordinate system.

From there you will adapt your Z buffering code to handle a wider range of depth variation without visual artifacts.

Then you will load both normal & emissive maps. Then use their data to influence the lighting equations from your shaders. This culminates with you loading an image-based lighting (IBL) cube map and using it to integrate a highly realistic PBR pixel shader.

### Task A1

Copy Work2 as your base, and switch to using the Bottle GLTF model. (Don’t forget to export it from Blender first)

A yellow bottle with a red lid

Description automatically generated

Note the text on the bottle is **backwards**… This is due to a major issue we have been avoiding up till this point.

***Hint:*** *Don’t forget to adjust the CMakeLists.txt to correctly name the project after copying your source over.*

### Task A2

We are currently using a left-handed coordinate system, whereas GLTF uses a right-handed coordinate system. We have two choices here: Conform our coordinate system to GLTF’s system or adjust the incoming matrix/transform data and geometric winding to account for the difference.

A black object with blue and green arrows

Description automatically generated

**The simplest way to make a GLTF model a left-handed model is to scale it -1 on the Z axis.** (see above image)

Now is a good time to add support for world matrices to the project via storage buffers. Alternately, you can add one world matrix to the **SHADER\_VARS** before sending it to the shader as a temporary solution. (however, be aware this will only work for a single mesh and limits the complexity of loadable models)

### Task A3

Regardless of which method you choose, create a World Matrix that flips the 3D model around the **Z axis**. Apply this new matrix by adjusting your vertex shader. After scaling on the -Z axis, the model will look like this:

A yellow bottle with a red lid

Description automatically generated

The final issue is that (as mentioned in the previous image) the GLTF coordinate system causes positive rotation to be considered counterclockwise. This means that based on our current **rasterizer** settings, only clockwise triangles will be rasterized.

### Task A4

To fix this, we must find where the front face setting is selected and set it counterclockwise, so Vulkan knows to render those faces instead of the other way around. (this is part of the rasterization pipeline settings)

A yellow and black object with a red logo

Description automatically generated

Once you incorporate the -Z world matrix and culling mode swap, the model should look correct. Don’t forget to also move multiply the vertex normal by the new world matrix. If you don’t, the lighting will start acting strangely. You may see artifacts like specular highlights in the shaded areas of the model where they don’t belong.

### Task B1

Change the near plane so it is at **0.00001f** and the far plane to **10000.0f**

Move the camera away from the bottle and you should start to see the Z buffer’s accuracy start to break down:

A yellow bottle with a black lid

Description automatically generated

Why this happens and how we can improve it: <https://developer.nvidia.com/content/depth-precision-visualized>

### Task B2

It’s fair to say these are extremely aggressive distance thresholds we are asking from our Z Buffer (Particularly the Near Plane). And while a single bottle doesn’t need this level of depth range, a giant 3D model representing an entire game level/world probably would.

The good news is we can get a more usable curve to our depth buffer precision by reversing our Z depth:

1. **Clear** the Z buffer to 0 not 1. (0 is now the Far Plane)
2. In the **Depth Stencil State** swap the min/max Depth Bounds. (0 ⬄ 1)
3. Switch the **Z comparison mode** to GREATER instead of LESS\_THAN.
4. In the **Viewport**, swap the min/max Depth values. (0 ⬄ 1)
   1. Swapping the Near/Far of the projection matrix can achieve the same result if required.



### Task B3

Use your understanding of vector/matrix math to **rotate** the light’s **direction** around the model slowly over time:



I did this by tracking the elapsed seconds since the start of the program. I then used that time to **rotate** a temp matrix on the **Y axis** by interpreting time as radians. I then rotated the sun’s original direction by that matrix.

Our project is fully set up for rendering precise GLTF geometry. Now let’s incorporate those remaining textures!

### task C1

Load the **normal** and **emissive** maps from the first material in the GLTF file.

A screenshot of a computer

Description automatically generated

You can see the normal map is mostly flat (Purple) but there are some spots where the normal data seems to indicate that the logo has been embossed into the metal of the bottle. Also note the surface of the dark grey metal has more normal variation which indicates it is less smooth.

A screenshot of a computer

Description automatically generated

The emissive map is mostly empty unused space. However, if you zoom in you can see there is a small section with some numbers/writing. This seems to be a glowing LED indicator built into the bottle. This appears to be the only part of the bottle that emits its own light/glows.

### task C2

In the **Vertex Shader**, multiply the incoming **tangent vector** into World Space.

**IMPORTANT:** The tangent vector is a **direction** not a position. The **W** component is used to determine the polarity of the tangent relative to the surface based on the UV data. It’s essentially always just 1 or -1 and gets used later.

This means you must preserve the value of **W**, but it cannot affect the transformation of the 3D vector/direction. Treat the tangent like you would a vertex normal, then restore the **W** and send it to the **Fragment Shader**.

### task C3

In the **Fragment Shader**, normalize the incoming tangent vector (XYZ only). Compute a new 3D vector called the **binormal** by crossing the surface normal with the tangent vector (Both in World Space).

Scale this new **binormal** by the tangent’s **W** value you preserved from the Vertex Shader. (Flipping it if required)

### task C4

We now have everything we need to create something called the **Tangent Binormal Normal** matrix. (TBN)

This is a **3x3 Matrix** used to transform any normal we read from the normal map into the **World Space** of the triangle we are currently rasterizing.

Create the **TBN** matrix like so:

ROW 1 = [Tangent]

ROW 2 = [Binormal]

ROW 3 = [Normal]

***NOTE:*** *You may wonder why we don’t just store tangents in object/world space to begin with. This would be fine for static 3D models; however, many 3D models have animations that distort the placements of triangles over time. By keeping the normal data relative to the triangles, we account for any dynamic changes to the overall topology.*

### task C5

Now that we have our TBN matrix, we can use it to calculate a significantly more detailed per-pixel normal found in the normal map texture. Retrieve the final surface normal like so:

1. **Sample** the **XYZ color** from the Normal Map at the current UV location.
2. **Scale** the normal map color by **2x** so it ranges from **0.0f – 2.0f** (instead of 0.0f – 1.0f).
3. **Subtract 1.0f** from all components of the resulting vector so the range shifts to **-1.0f to +1.0f.**
4. Use the **TBN** matrix to transform the resulting triangle localized normal into **World Space**.

Once you have this new pixel-based normal vector, use it in all your lighting and shading calculations:



Once you incorporate the normal derived from the normal map in the directional lighting algorithm (instead of the existing vertex normal). You should see a rather stark change in lighting/surface detail.



If you slow down the light rotation & study the behavior, you will notice that the top of the letters in the normal map are not being lit up. This is an issue because the green channel in the normal map indicates that those surfaces should be getting lit up by our light shining from above when it passes over them:

A close up of a name

Description automatically generated

This happens because the normal maps used in **GLTF models** are OpenGL style normal maps. In these maps, **green means up** or **+Y** in the normal map. In DirectX style normal maps, **green instead means down** or **-Y**. Again, this difference is mainly due to the right-handed vs. left-handed coordinate systems where these Normals will be used.

**Adjusting for this is quite simple:**

Just invert (not negate) the value of the **green channel** after it is loaded from the normal map texture but before you shift its range and multiply it by the **TBN** matrix. (ex: green = 1.0 – green)



Once you do this, you should see the tops of the letters and other raised/embedded surfaces properly react to the directionality of the light source. As you can see, normal mapping makes a pretty big difference in the perceived detail/complexity of a surface. (As long as the user doesn’t look too closely of course!)

### task C6

Use the **emissive map** to add light/glow to the surface. Thankfully this is much easier to integrate than the normal map was. Just load the texel from the emissive map and simply **add** it to the final formula that combines all the other aspects of the light & material:

A close-up of a clock

Description automatically generated

Our LED display embedded behind glass. Note how it continues to glow even when the light does not hit it directly.

## section D | Physically based rendering (PBR SHADERS)

#### OVERVIEW

We should have all the tools needed to integrate such a shader into our code base. For this last section we will be loading three special textures and then switching over to a pre-written PBR shader.

If you wish to learn more about PBR in greater depth, I suggest reading these articles:

<http://www.codinglabs.net/article_physically_based_rendering.aspx>

<http://www.codinglabs.net/article_physically_based_rendering_cook_torrance.aspx>

### task D1

From the provided PBR IBL ENV folder, load the **lut\_ggx.png** texture file into your existing texture array:

A screenshot of a computer

Description automatically generated

The **PBR shader** will use this to help calculate the proper split-sum approximation for Cook-Torrence specular.

### task D2

Ok… one down two more to go. The next two files are not your normal run of the mill texture formats. They are something known as **cube maps** and are 6 images forming a hollow box. Most standard image formats cannot represent this style of texture. (BTW, this is also the type of format used for a SkyBox/Environment map in games)

To Load these textures, you will need to link the **ktx** texturing library included in this assignment. Inside the **ktx** folder you will find **README.txt** that explains how to link the library into your project/CMakeLists.txt and what headers to include in your code.

Once you do this. You should be able to **include & compile** the provided **TextureUtilsKTX.h** header file.

### task D3

With the library code working, let’s try it out by loading the **diffuse.ktx2** cube map image using our new utility function in **TextureUtilsKTX.h**. Load the file and add it to the existing texture vector you have already been using:

A screenshot of a computer

Description automatically generated

Make note of the fact that this texture has multiple slices and a **16bit** floating point format per channel. Try clicking on the slice/face drop down menu in the top-left corner to see the different **cube faces**.

Even though this is technically a different type of texture (a cube map) it can still be loaded into the main texture array we already have. Normally we would need to go mess with descriptor sets, but thankfully we are using **BINDLESS** texturing which lets us bypass that previous requirement. (love bindless!)

### task D4

Same deal as before. Load the **specular.ktx2** cube map texture and verify its presence using RenderDoc:

A screenshot of a computer

Description automatically generated

Notice that the Specular **IBL** (Image Based Lighting) texture in PBR is much more legible than the diffuse version. This makes sense since this cube map is the source of things like clear metallic reflections vs. diffused/scattered ambient light/energy.

However, this only stays crystal clear for the first Mip level [0]. If you check the lower levels, you will see how the sampled reflection gets more and more scattered. This is because something like brushed metal still reflects its surroundings but due to the roughness of the surface, reflected rays are not always parallel in proximity.

### task D5

Ok… we have everything we need. Switch your pipeline over to using the **FragmentShader\_PBR.hlsl** shader. This shader has been fully set up to work with all the data you have recently loaded and are sending from the vertex shader. It is very likely that it won’t work immediately without a few tweaks since our code may be a bit different.

Take careful note of which textures are in your texture array vs. the **order** I am using them in the shader. You may also need to adjust the incoming vertex format and SEMANTICS to match whatever your **Vertex Shader** is currently sending as **output** to the Pixel/Fragment stage:

A green bottle with red cap

Description automatically generated

A close up of a bottle

Description automatically generated

Once you get the PBR shader running, the difference in quality should leap out at you. Metal should look like metal, plastic like plastic. Even the painted logos on the metal bottle will react differently than the bottle itself.

If you open the model in Blender and turn on the material view, you will notice it looks much closer to what you have now. It won’t be exactly the same (since Blender has multiple point lights to better illuminate the scene). It also uses a different IBL image by default. However, it will be MUCH closer than what you had with classic materials. If you want to see the *flexibility* of PBR, try some different test environments from the links in the **README** found in the PBR IBL ENV folder. (Watch how the bottle surface *adapts* to different surroundings)