1.1 Prep && Debug:

- **Shading Space:**

I make all light shadings in the world space. To convert all variables into the world space, I simply pass the light position and camera position in the fragment shader. And I calculate a TBN matrix in the vertex shader and pass it into the fragment shader to convert the normal from tangent space to the world space.

- **Descriptor Layout and bindings && Uniform Data**

1. Descriptor Set Layout = 0: view_DescriptorSetLayout,
    a) Binding = 0,

I pass a uniform buffer(view_UniformBuffer) in the vertex shader, which contains only a projection Camera Matrix in mat4, considering that I do shadings in the world space, I have to convert positions from local coordinates from model space to the world space. And here, Sponza never moves, so the model matrix(M) is always an identity matrix. And VP is the project camera matrix, So I just use it.

```
layout(set = 0, binding = 0) uniform MVP_MATRICES {
    mat4 VP;
}mvp;
```

2. Descriptor Set Layout = 1: pbrTexture_DescriptorSetLayout
    a) Binding = 0: albedo texture
    b) Binding = 1: metallic texture
    c) Binding = 2: roughness texture
    d) Binding = 3: normal texture
    e) Binding = 4: ambient occlusion texture

All of them are essential textures for CW02 PBR shading.

```
layout(set = 1, binding = 0) uniform sampler2D albedoMap;
layout(set = 1, binding = 1) uniform sampler2D metallicMap;
layout(set = 1, binding = 2) uniform sampler2D roughnessMap;
layout(set = 1, binding = 3) uniform sampler2D normalMap;
layout(set = 1, binding = 4) uniform sampler2D aoMap;
```

3. Descriptor Set Layout = 2: light_DescriptorSetLayout
    a) Binding = 0

I pass another uniform buffer(light_UniformBuffer) in the fragment shader, which contains a light position (vec4), a light color (vec3), and a view pos(vec4). Where the light position and light color are variables for lighting shading, and camera position is camera position. Pass them in the fragment shader, to keep variables are accualte in the shading process. And vulkan likes meomery align, so the structure in the fragment shader looks as follows:

```
layout(set = 2, binding = 0) uniform LIGHT {
    vec4  lightPos;
    vec3  color;
    float padding;
    vec4  viewPos;
}light;
```

1.2 Lighting

1.2.1 Global Sponza View

1.2.2 Specular BRDF:

D (normal distribution):

I used two diferent equation, the left one picture uses Disney's reparameterization with the equation as follows:

$$D_{GTR} = c/(\alpha^2 cos^2\theta_h + sin^2\theta_h)^\gamma$$

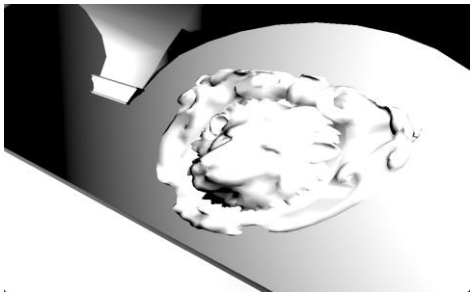
And the right picture uses the equation in the cw02 description. Compare with these two results in specular BRDF, I choose to use the first equation.

F (Fresnel Term):

I use the equation in the cw02 description. It is what I want to implement.

G (Masking Term):

I implement Cook-Torrance model in the cw02 description.



PBR Specular Item:

With lighting, the lion head looks pretty good.

Diffuse BRDF:

I implemented the Lambertian model, where the lion's head is totally black, and diffuse BRDF did not provide any contribution to the lion's head.

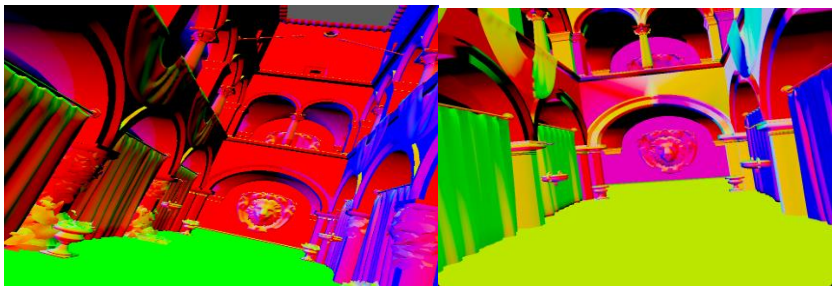
All of them get the result I want to implement, and the result looks good.

1.3 Alpha Masking

I implement two rendering pipelines. One is for "normal" shading, except AO, and one is for AO mapping only. The following three pictures displayed, rendering both pipelines, rendering pipeline without AO, and rendering AO only respectively.

1.4 Normal mapping

Normal visualization: before mapping (left) && after mapping(right):



When I first get the normal visualization after normal mapping, I thought the result looks so weird, because the floor looks yellow, but the left curtain looks green. Then, I found the reason that I use the texture format in R8G8B8A8_SRGB in the program. According to the description in the Vulkan documents and email from teacher, I use R8G8B8A8_UNORM instead ,and make a gamma correction in the end of the shader.

Color visualization: before mapping (left) && after mapping(right):

In the folds of this fabric, there is a higher concentration of normals, which allows for a better visualization of the depth differences. Clearly, after applying normal mapping, the details of the shadows in the folds can be rendered accurately.

As for tgen, I include it into 1.5, while I revise the code a lot to make it good to use in CW02.

1.5 Mesh data optimizations

To implement normal mapping by constructing a TBN matrix, I carried out the following steps:

1. Modify the tgen library for easier use:
   a) The tgen library provides four C-style functions, which require extra operations for point arrays and glm vectors or quaternions. I rewrote the code using glm API, including glm::vec3, glm::vec4, glm::quat, and other linear algorithms. I did this not only for data structure efficiency but also for

safety. When using the "normalize" function in tgen, it sometimes generates NaN results due to floating-point precision. Referring to the "NormalizeSafe" function in Assimp (Open Asset Import Library) on GitHub, I added extra operations to ensure the result would not be an invalid number, where eps = 1e - 8.

```
for (uint32_t i = 0; i < numUVVertices; ++i) {
  if (glm::length(vTangents3D[i]) > eps)
    glm::normalize(vTangents3D[i]);
  if (glm::length(vBitangents3D[i]) > eps)
     glm::normalize(vBitangents3D[i]);
}
```

b) Meanwhile, the tgen library uses a vec4 to record tangents, with the last value being the sign. I think this is a redundant operation, so I simply flipped the sign when the sign < 0. Thus, I directly computed a tangent in vec3. The last position is unnecessary. Bake TBN Quaternion in the baking process. While I could compute tangent in vec3, I could simply get the TBN matrix in mat3 and convert it into a glm::quat. Instead of passing normal and tangents in the shader. I just pass a glm::quat in vec4, and calculate it to a mat in the shader.

2. Bake TBN Quaternion during the baking process. Since I could compute the tangent in vec3, I could easily obtain the TBN matrix in mat3 and convert it into a glm::quat. Instead of passing normals and tangents in the shader, I just passed a glm::quat in vec4 and calculated it as a mat in the shader.
3. Defer the compression process to the loading process. I tried to compress a quaternion into a 32-bit integer using the R32_UINT format. This is similar to the R10G10B10A2 encoding. I didn't include it in the baking process because the result didn't look as good as passing a quaternion in. It lost many details, especially in the fold details, where normals changed significantly, and 9-Bit was not enough to show the most accurate details.
4. I passed the original TBN quaternions in 4floats and compressed TBN quaternions in 1uint32 simultaneously to compare the differences between both. Based on my preferences, I would not use compressed data if I wanted to achieve the best visual effects.

Other Tips:

Camera operation:

Mouse Left and Move: Yaw and Pitch

Mouse Middle Button: Hide/ Display Cursor

Mouse Right and Move: Translate in X and Y axis.

Mouse Scrolling: Move forward and backward.

Manual pipeline switch:

Due to I have not implement dynamic pipeline switch in runtime, switch uncompressed/compressed pipelines and withAO/withoutAO, you have to comment some codes in shader:

Uncompressed or compressed TBN quat in pbr.vert:

Uncompressed situation, use `tbn = quaternionToTBNMatrix(inTBNQuat);`

Compressed situation, use `tbn = quaternionToTBNMatrix(decode_quaternion());`

enable AO in ao.frag: `outColor = vec4(texColor);.` To disable AO, just comment this line.