COMP5822M Coursework 2

Coursework 2 aims at implementing a physically-inspired lighting model for rendering. It further refines the shading using the alpha masking and normal mapping techniques and finally optimizes the mesh data for better performance.

Coursework 2 includes two applications, a non-graphical tool that reads an OBJ file and bakes it into a simple binary format and a renderer. The coursework is split into five tasks and this report will give the details and strategies used for their implementation.

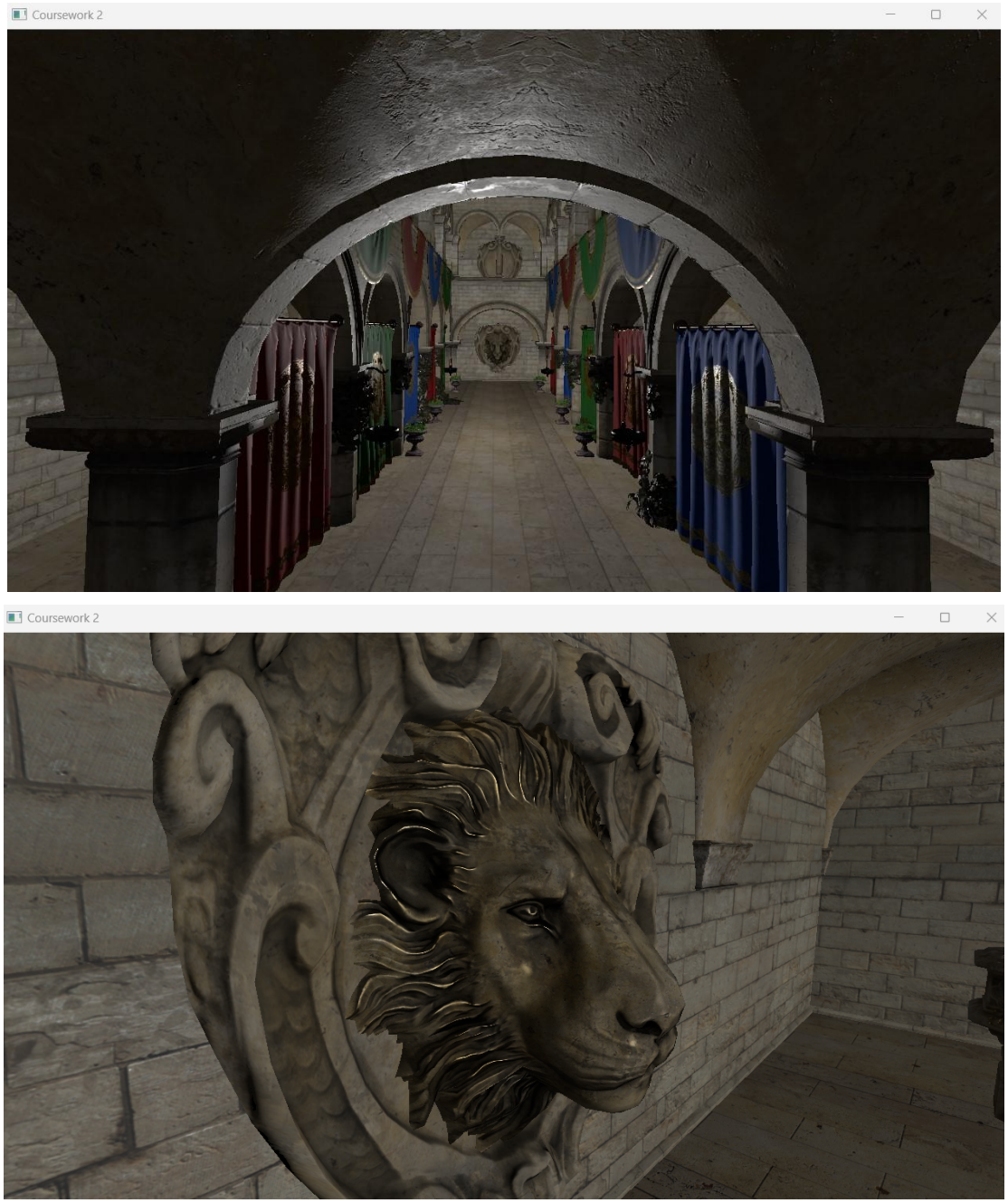


Figure 1: Final rendered images from Coursework 2

Tasks

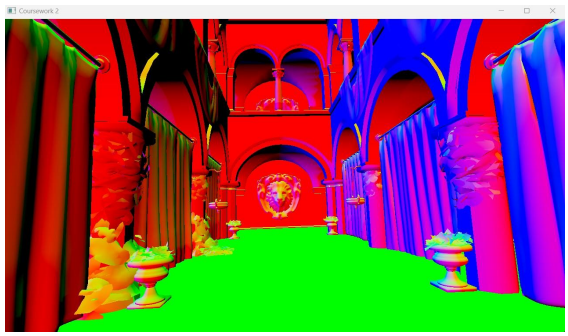
1. Prep and Debug

The first task involves setting up a Vulkan renderer that loads and draws the binary file produced by the baking step. All shading is done per fragment in world space as recommended. The details of the implementation are as follows:

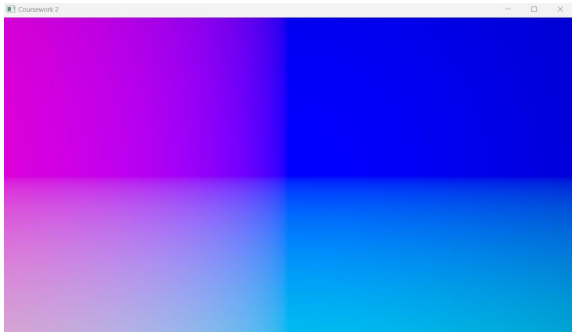
- **Shading Space:** World Space
- **Descriptor set layout and Descriptor bindings:**
  The implementation has three Descriptor layouts:
  1. *sceneLayout*: This layout is for the global scene uniforms and is of type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER. It is set Binding0 of Set0. This layout is used to hold SceneUniform data and is bound once per frame during rendering. It is set to be accessible from both vertex and fragment shader
  2. t*exturedobjectLayout*: This layout is for the descriptors that store the various texture data for the meshes and contains four bindings of type *VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER* for each of the baseColor, roughness, metalness and normalMap textures of a given material. The rendering is done by iterating through materials and drawing all the meshes that use the same material. Hence, these descriptors are bound once per material during rendering. These are set to be accessible only in the fragment shader.
  3. *alphamaskedobjectLayout*: This layout is similar to the texturedobjectLayout with one additional binding for alphaMask texture. These descriptors are also bound once per material.

- **Uniform Input Data.:**
  The scene uniform inputs are passed to the shaders as buffers. The Uniform Buffer Object holds the data from the *SceneUniform* struct. It contains the transformation matrices and also camera and light data required for rendering. It is packed following the std140 layout rules.

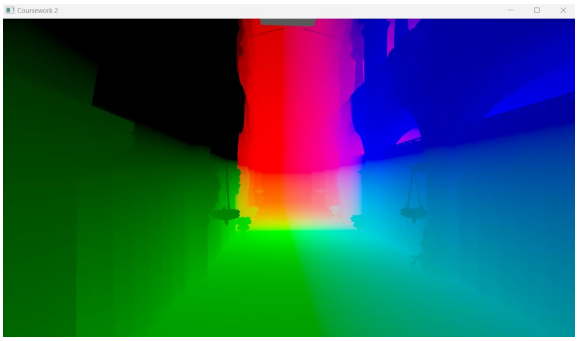
```
struct SceneUniform
{
        glm::mat4 camera;
        glm::mat4 projection;
        glm::mat4 projCam;

        glm::vec3 cameraPosition;
        alignas(16) glm::vec3 lightPosition;
        alignas(16) glm::vec3 lightColor{ 1.0f, 1.0f, 1.0f};
        alignas(16) glm::vec3 ambientColor { 0.02f, 0.02f, 0.02f };
};
```



(a) Normal



(b) View direction



(c) Light direction

Figure 2: Visualization of normalized normal vector(a), view direction(b) and light direction(c) with their world space values.

2. Lighting

This task involves implementing a physically-inspired lighting model for rendering which consists of a Lambertian diffuse component and a specular component based on a microfacet BRDF using the Blinn-Phong normal distribution function. The model is implemented with a single point light source with color (1,1,1) and an ambient contribution of 0.02. The light position is animated to orbit around the z-axis that can be controlled by pressing the spacebar.

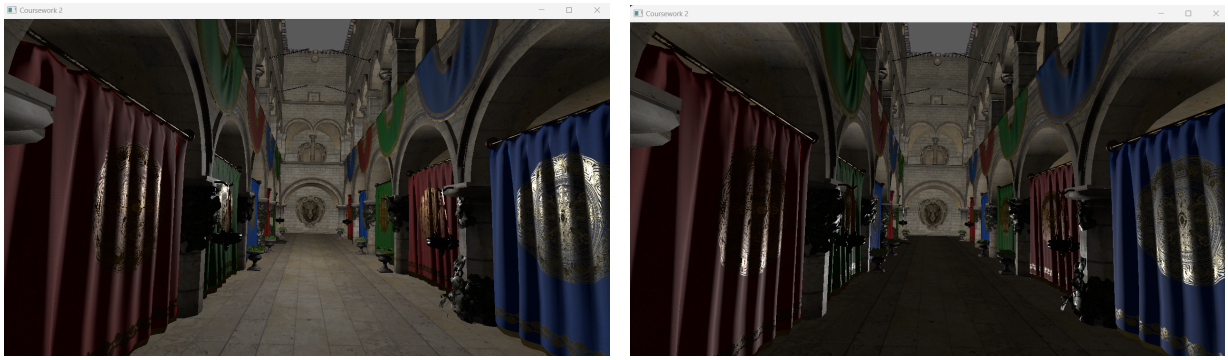


Figure 3: Results of lighting implementation with different light positions

Figure 4: Results of lighting implementation on the Lion Head metallic sculpture

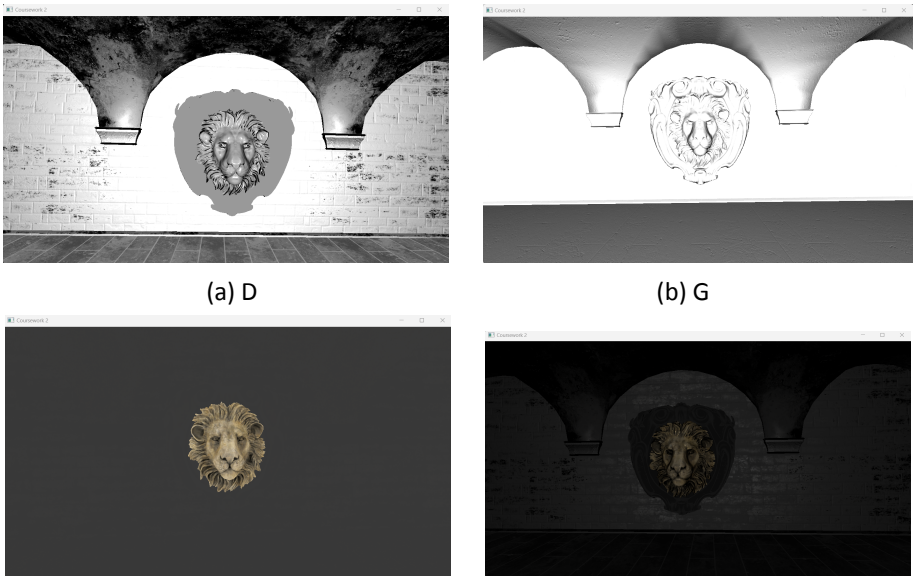
The Lambertian diffuse term accounts for the diffuse reflection of light, assuming that incident light is scattered equally in all directions around the surface normal. This would mean that the light would not cause specular

reflection on the surface and hence can be particularly useful for modeling rough, non-specular materials. The Lambertian diffuse term of a metallic object would be very low or close to zero as most of the light falling on a metallic surface would be reflected specularly. In Figure 5, we can see the metallic Lion head sculpture has zero or very negligible diffuse component giving it a black color.

Figure 5: Visualization of Lambertian diffuse term

The PBR Specular Term determines the intensity and color of the specular highlights based on the surface properties. The specular term will be stronger and more focused on a smooth, metallic surface than on rough surfaces. The calculation of specular term has the contributions from the following factors and it is visualized in Figure 5 :

- normal distribution function, D: It evaluates the fraction of microfacets oriented such that it reflects the light in the viewing direction. In a rough surface, the microfacets will be oriented in different directions resulting in reflection taking place in many directions and spreading the incoming light. A smooth surface, on the other hand, will have a narrower range of angles, resulting in a more focused specular reflection.
- masking function, G: accounts for attenuation of reflected light because of the masking of microfacets due to the geometry of the microsurface. With the random arrangement of microfacets on a rough surface, the G term decreases, as more fraction of the reflected light gets blocked by the uneven geometry.
- Fresnel term, F : describes the fraction of light that gets reflected off a surface based on the angle of incidence and the refractive indices of the materials. The light is reflected by the metallic objects like the lion head while the wall being rough does not reflect so much.



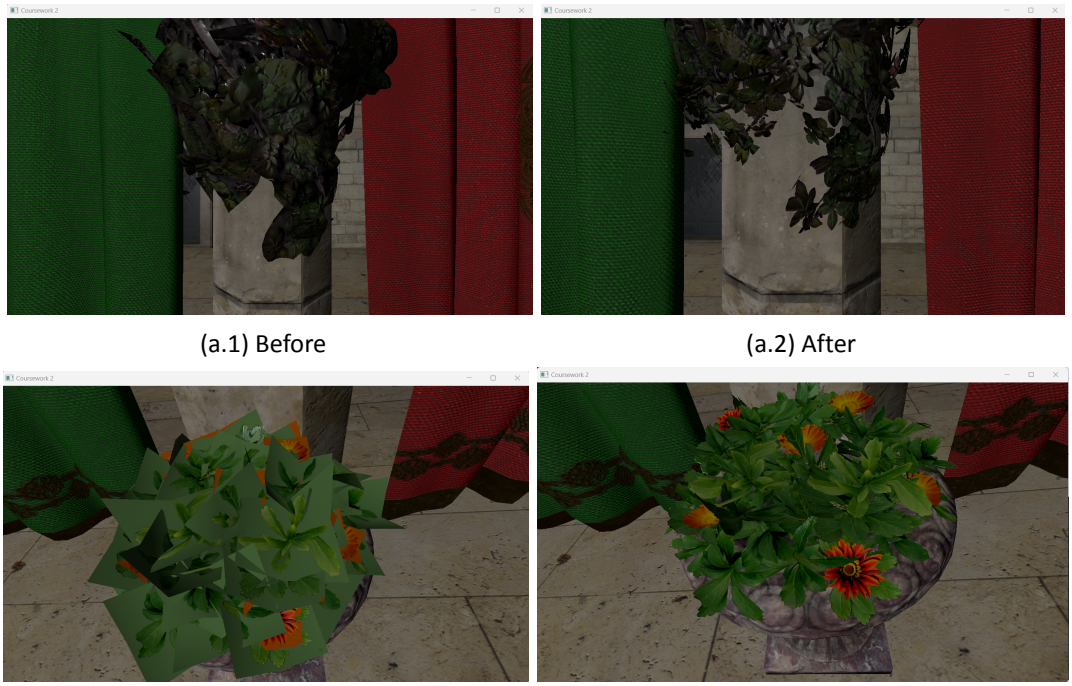
(a) D (b) G

(c) F (d) PBR specular term

Figure 6: Visualization of factors that contribute to the Specular term

3. Alpha Masking

For this task, a new graphics pipeline is added to render alpha masked textures. The pipeline uses the same vertex shader as the default pipeline and alpha masking is done in the fragment shader. This refined objects like the foliage and chains in the Sponza scene.



(a.1) Before (a.2) After

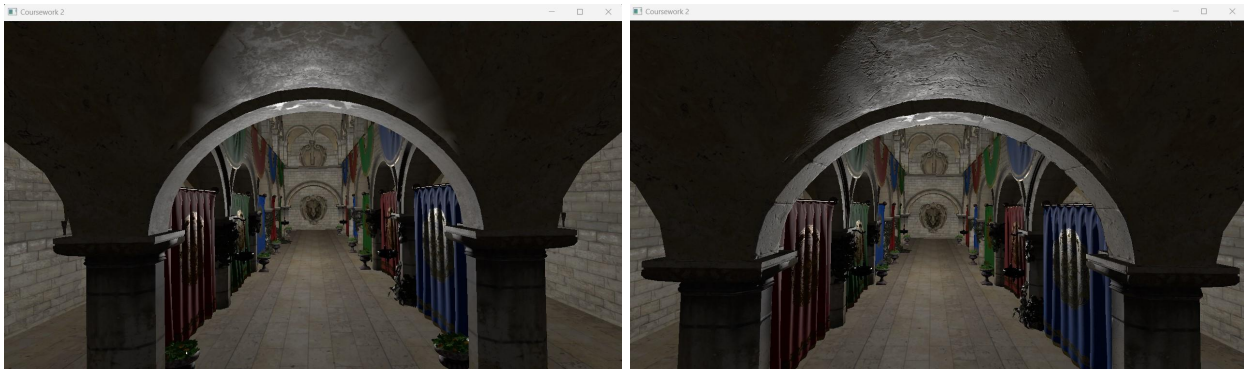
(b.1) Before (b.2) After

Figure 7: Results of alpha masking implementation on foliage

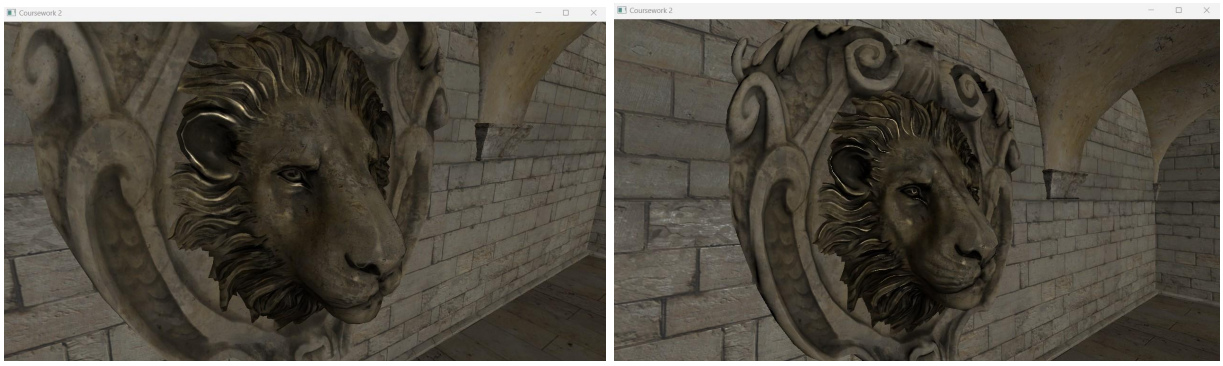
4. Normal mapping

This task involves improving details of the rendering using a normal map. For this, the tgen library is used to compute the tangents. tgen produces a vec4 tangent where the w component indicates the direction of the tangent basis. w = 1 indicates that the tangent basis is right-handed and w = -1 indicates that it is left-handed.

Normal mapping uses the tangent space of the surface to orient the surface normals. The tangent basis may abruptly flip direction between triangles sometimes and this can cause mirroring in the rendered image as this would cause the normals to flip direction as well. Hence, the sign of the w component of the tangent can be used to determine if the surface normal needs to be flipped. If the sign is negative, the normal is flipped, else it is left as is. This ensures that the surface normals are oriented correctly .

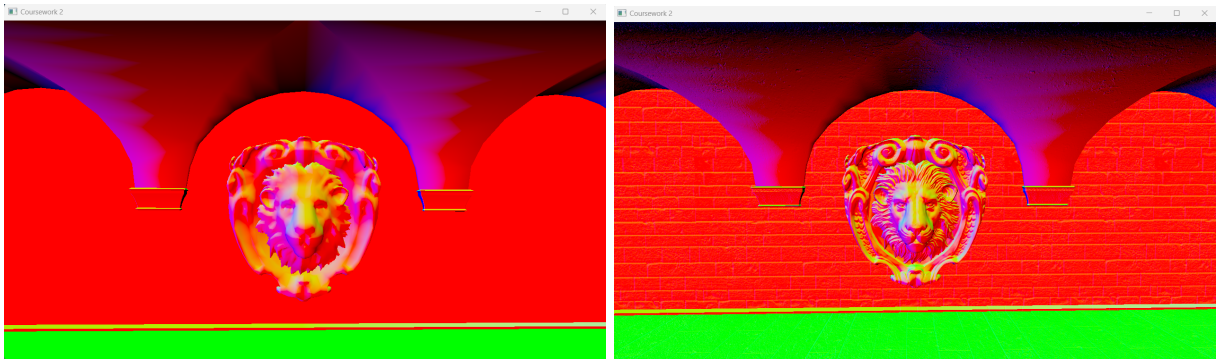


(a.1) Before (a.2) After



(b.1) Before (b.2) After

Figure 8: Results of using normal map for rendering



(b.1) Before (b.2) After

Figure 9: Visualization of normals before and after using normal maps for rendering

5. Mesh data optimizations

The final task involves optimizing the mesh data, particularly the normal and tangent data. The tangent space vectors form an orthonormal TBN coordinate frame. This can express a rotation that will transform the orthonormal frame of tangent space into the standard frame in world space. Hence the TBN data can be encoded into a quaternion, reducing the tangent space representation to 4 components (16 bytes). Further, since the quaternions are normalized, there are only 3 degrees of freedom and hence, only the first three components of the quaternion need to be stored, bringing the size down to 12 bytes. The w can be calculated using the formula

$$w = \text{sqrt}(1 - x^2 - y^2 - z^2)$$

Finally, we can assume that 16 bit floats might be enough to represent components of a quaternion as they are within the range [-1,1]. The vec3 quaternion is discretized and is stored as 16-bit floats (*std::vector<std::vector<half>> tbnQuaternion;*) using the "half" library, reducing the TBN space size to 6 bytes.

NOTE: Implementation in progress

```
glm::vec3 TBN_to_Quaternion(glm::mat3 TBN)
{
    float t;
    glm::vec4 q;

    if (TBN[2][2] < 0) {
        if (TBN[0][0] > TBN[1][1])
        {
            t = 1 + TBN[0][0] - TBN[1][1] - TBN[2][2];
            q = glm::vec4(t, TBN[0][1] + TBN[1][0], TBN[2][0] + TBN[0][2], TBN[1][2] - TBN[2][1]);
        }
        else {
            t = 1 - TBN[0][0] + TBN[1][1] - TBN[2][2];
            q = glm::vec4(TBN[0][1] + TBN[1][0], t, TBN[1][2] + TBN[2][1], TBN[2][0] - TBN[0][2]);
        }
    }
    else {
        if (TBN[0][0] < -TBN[1][1]) {
            t = 1 - TBN[0][0] - TBN[1][1] + TBN[2][2];
            q = glm::vec4(TBN[2][0] + TBN[0][2], TBN[1][2] + TBN[2][1], t, TBN[0][1] - TBN[1][0]);
        }
        else {
            t = 1 + TBN[0][0] + TBN[1][1] + TBN[2][2];
            q = glm::vec4(TBN[1][2] - TBN[2][1], TBN[2][0] - TBN[0][2], TBN[0][1] - TBN[1][0], t);
        }
    }
    q *= 0.5 / sqrt(t);

    return glm::vec3(q.x, q.y, q.z);
}

void Generate_TBN_Quaternion(IndexedMesh& indexedMesh)
{
    assert(indexedMesh.tangent.size() != 0);

    for (unsigned int i = 0; i < indexedMesh.vert.size(); i++)
    {
        glm::vec3 normal = normalize(indexedMesh.norm[i]);
        glm::vec4 tangent = normalize(indexedMesh.tangent[i]);
        glm::vec3 bitangent = normalize(glm::cross(normal, glm::vec3(tangent.x,tangent.y,tangent.z)) * tangent.w);

        glm::vec3 quaternion = TBN_to_Quaternion(glm::mat3(tangent, bitangent, normal));

        indexedMesh.tbnQuaternion.push_back({ half(quaternion.x), half(quaternion.y), half(quaternion.z)});
    }
}
```

Figure 10: Code snippet for encoding TBN to three 16 bit floats in main.cpp of cw2-bake

References:

1. https://vkguide.dev
2. https://vulkan-tutorial.com
3. https://physicsforgames.blogspot.com/2010/03/quaternion-tricks.html
4. https://math.stackexchange.com/questions/893984/conversion-of-rotation-matrix-to-quaternion
5. https://boksajak.github.io/files/CrashCourseBRDF.pdf