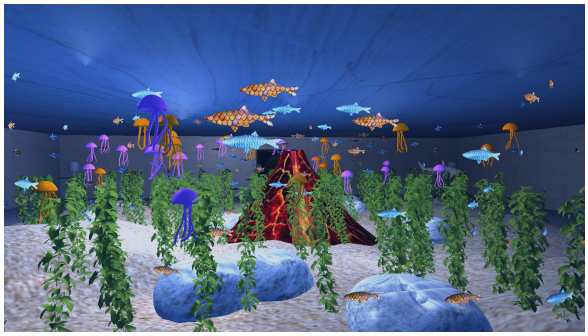
CS7GV6 Computer Graphics
Project Deliverable 3
Aquarium Display

Name: Michael Millard
Student ID: 24364218

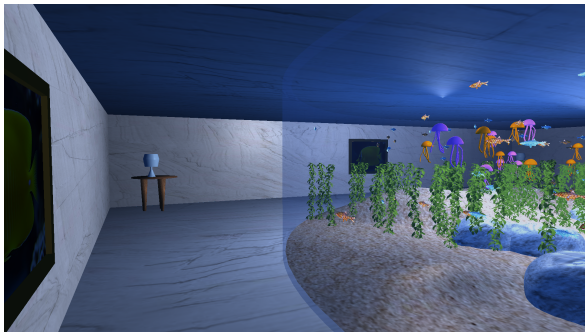
December 16, 2024

Scene Description

This is my final submission (deliverable 3) for CS7GV6 (Computer Graphics) 2024. My project was entitled "Aquarium Display" and is set indoors with a large, cylindrical aquarium in the centre of a square room. The outer walls, roof and floor are styled in white quarts with tables and table lamps placed in each of the four corners of the room. There are also four pictures of a fish hung around the room, one on each wall. Within the tank itself, there are two differently textured fish models, a shark model, many kelp models (which are animated hierarchically), rock models, a self-made terrain styled with sand, a volcano in the centre, many jellyfish bobbing up in down (orange and purple ones), and 5 point lights within the tank. Figures 1a and 1b below illustrate the description of the aquarium scene provided above.



(a) View of the scene from the camera spawn point



(b) View of the room surrounding the tank

Figure 1: Two views of the aquarium display scene

Lighting and Shading

In this assignment I implemented Phong Illumination to achieve realistic underwater lighting effects with multiple light sources. I had 5 point light sources inside the aquarium. One in the centre of the aquarium on the roof, and 4 others surrounding it. These lights are shown in Figure 2 below; although they are faint as this is how I desired to style the scene.

Figure 2: The 5 point lights in my aquarium

In order to implement these point lights, I needed to create a point light structure in my fragment shader, populate instances of this structure for each point light and then accumulate the lighting effects

by looping through each source to compute the combined ambient, specular and diffuse lighting values. The structure is shown below in Figure 3.

```
struct PointLight
{
    vec3 position;  // Position of the light
    vec3 ambient;   // Ambient color intensity
    vec3 diffuse;   // Diffuse color intensity
    vec3 specular;  // Specular color intensity
    float constant; // Attenuation constant term
    float linear;   // Attenuation linear term
    float quadratic;// Attenuation quadratic term
};
```

Figure 3: Point light structure

As explained above, I then looped through each light source and computed the combined effects of these sources in the fragment shader. This loop implementation is shown below in Figure 4.

```
    // Init to vec3s of 0
    vec3 ambient = vec3(0.0);
    vec3 diffuse = vec3(0.0);
    vec3 specular = vec3(0.0);

    // Loop through point lights
    for (int i = 0; i < 5; i++)
    {
        // Ambient lighting
        ambient += pointLights[i].ambient;

        // Diffuse lighting
        vec3 norm = normalize(normal);
        vec3 lightDirection = normalize(pointLights[i].position - fragPos);
        float diff = max(dot(norm, lightDirection), 0.0);
        diffuse += pointLights[i].diffuse * diff;

        // Specular lighting
        vec3 viewDirection = normalize(viewPosition - fragPos);
        vec3 reflectDirection = reflect(-lightDirection, norm);
        float spec = pow(max(dot(viewDirection, reflectDirection), 0.0), specularExponent);
        specular += pointLights[i].specular * spec;

        // Attenuation
        float distance = length(pointLights[i].position - fragPos);
        float attenuation = 1.0 / (pointLights[i].constant + pointLights[i].linear * distance
        + pointLights[i].quadratic * (distance * distance));

        // Apply attenuation to the light contributions
        ambient *= attenuation;
        diffuse *= attenuation;
        specular *= attenuation;
    }

    vec3 lighting = ambient + diffuse + specular;
```

Figure 4: Looping through point lights in fragment shader

The vertex shader had a very simple implementation. It received the vertex position, normal and texture coordinates as its inputs; it had model, view and projection matrix uniforms that got set at the beginning of each iteration of the render loop; and its outputs were the fragment position, the normal

vector, and the texture coordinates.

Since I had a glass object (the fish tank), one of my greatest challenges was implementing this in the fragment shader. I initially attempted to import a glass texture I had designed in Blender with the cylindrical fish tank (see Figure 8 below). However, this proved incredibly difficult and so I opted to rather have a uniform boolean that I could set in the fragment shader when drawing my models.

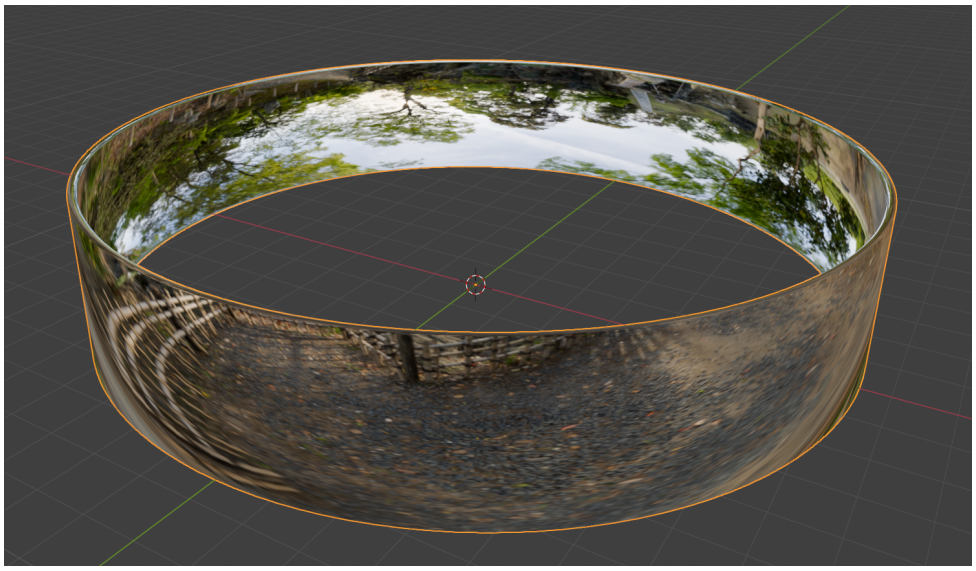


Figure 5: My glass fish tank 3D model made in Blender

When I would be drawing my glass model, I would set the boolean variable called *useTexture* to false and draw the model with a fixed glass colour (light blue with a low alpha value of 0.2). Else, for all other models with textures I would set it to true and use the texture coordinates and diffuse texture. See the implementation of this below in Figure 6.

```
// For textures
if (useTexture)
{
    vec3 texColor = texture(textureDiffuse1, texCoords).rgb;
    fragColor = vec4(lighting * texColor, 1.0);
}
// For glass
else
{
    fragColor = vec4(lighting * glassColor.rgb, glassColor.a);
}
```

Figure 6: Fragment shader output decided by *useTexture* boolean

Finding the right combination of diffuse, ambient and specular values for my point lights was achieved through trial and error. The same goes for the values of the constant, linear and quadratic terms in the attenuation formula. In the end I found that the following set of parameters achieved a desirably lit aquarium, as provided below in Figure 7.

```
ambient = glm::vec3(0.1f, 0.2f, 0.4f);
diffuse =  glm::vec3(0.8f, 0.8f, 0.8f);
specular = glm::vec3(0.5f, 0.5f, 0.5f);
constant = 0.9f;
linear = 0.04f;
quadratic = 0.01f;
```

Figure 7: The ideal set of point light parameters for my scene

Not included in this list is the uniform float for the specular exponent, for which I found a value of 32 achieved a desirable level of shinyness.

Texture Mapping

For this submission I decided to design all of my models to only have diffuse textures for simplicity's sake. As such, I would design my models in Blender, create a new material and apply it to the object (all jpeg image textures), adjust the UV mapping to my liking to ensure the model looked as realistic as possible, before exporting from Blender and importing into OpenGL using the ASSIMP framework. An example of this UV mapping done for my volcano model in Blender is shown below in Figure **??**.

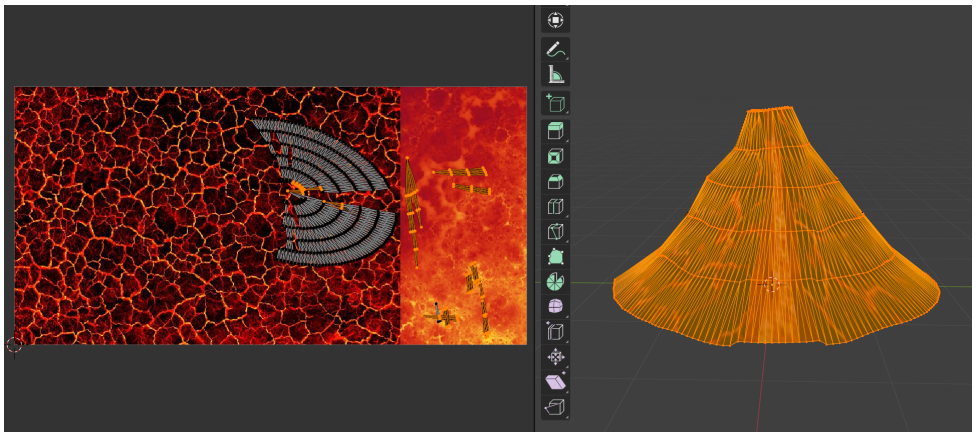


Figure 8: UV Mapping for my volcano 3D model done in Blender

The sub-figures below in Figure 10 shows a handful of the other 3D models contained within my scene, all of which were exported from Blender and imported into OpenGL using ASSIMP. The order of the models is as follows: an orange jellyfish, a volcano, sand terrain for the bottom of the tank, a table and lamp, a fish and a shark.

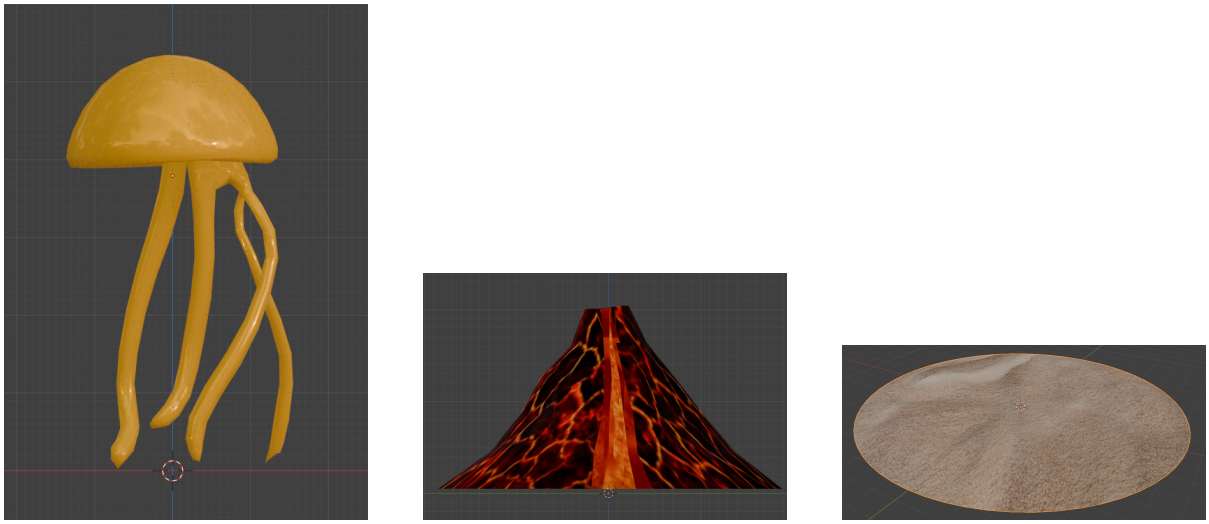


Figure 10: Six other models created and texture mapped in Blender

Advanced Features

My advanced features for this project included incorporating a glass fish tank into the scene, adding a shark feeding feature, and adding collision detection functionality to my application. The glass fish tank has already been discussed above in depth in my shader discussion and as such I will be focussing on the shark feeding feature here.

My shark model is initially animated to swim around the tank in a circle. It's movement is governed by the amount of elapsed time in the program and the standard library sinusoid functions, as shown below in Figure 11.

```
// Update shark pose params
float sharkRad = sharkModel.meshes[0].initRad;
float theta = elapsedTime * 0.1f;
sharkModel.meshes[0].mesh6DoF[tX] = sharkRad * cos(theta);
sharkModel.meshes[0].mesh6DoF[tZ] = sharkRad * sin(theta);
sharkModel.meshes[0].mesh6DoF[rY] = (float(M_PI) / 2.0f) - theta + float(M_PI);

for (unsigned int j = 0; j < static_cast<unsigned int>(sharkModel.meshes.size()); j++)
{
    sharkModel.meshes[j].mesh6DoF[rY] = sharkModel.meshes[0].mesh6DoF[rY] + 0.1f * sin(elapsedTi
    sharkModel.meshes[j].updateModelMatrix();

    // Set model matrix (multiply with previous model matrix for hierarchy animation)
    model = sharkModel.meshes[j].meshMatrix;
    shader.setMat4("model", model);
    sharkModel.meshes[j].draw(shader);
}
```

Figure 11: Moving the shark around th tank in a circle

The first few lines are concerned with orienting the shark correctly. Each of my models is designed to have its own set of meshes that make up their own model matrix. This matrix is made up of the rotations and translations implemented by the models 6 degrees-of-freedom pose, a variable I called "mesh6DoF" in my code. I created an enum to allow for easy indexing of this float array (tX = translate in x-direction, etc). The for loop that follows is my implementation of making the shark "wag" whilst it swims. I use the mesh index to compute a sinusoidal phase shift which results in a different "sway" to the other meshes in the model and gives the appearance of the shark swaying through the water.

However, upon pressing the "F" key, a batch of food pellets drops from the roof of the acquarium and the shark changes its route to pursue the food. An example of this "smart" animation taking place is shown below in Figure 12 next to a picture of the pellets designed in Blender. Note that the pellets are dark in the figure but are on the left hand side of the image just above the kelp.

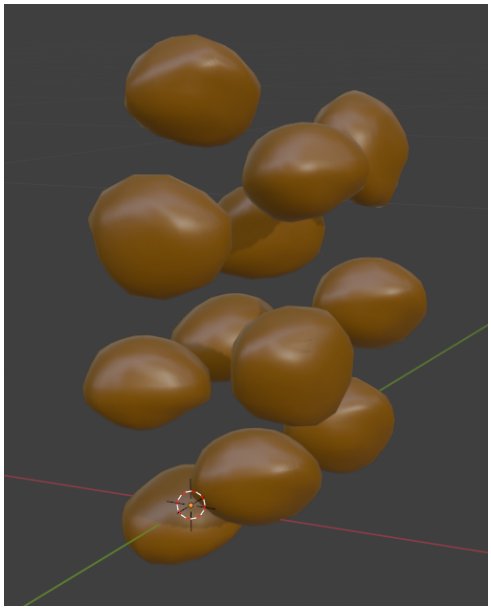


Figure 12: Shark pursuing food pellets

The food pellets would drop gradually over time and the shark would adjust its direction accordingly. The code implementing this food chasing animation is provided below in Figure 14. The effectiveness of this animation is hard to describe in a write up but there are two examples of this feature being shown in the YouTube video demonstration.

The code is provided on the next page due to spacial constraints. The first part of the code shows how the pellets fall from the roof over time (their y-coordinate gradually decreases). If the pellet goes below the floor before the shark reaches it, the animation ends.

The second part is more complicated but pertains to the shark reacting to the food. The direction vector to the food is first computed and the shark is gradually rotated and translated towards the food until it reaches it and eats it. This is checked by the magnitude of the vector between the food model and the shark model. Again, the wagging animation is added to the shark at the bottom.

```
for (unsigned int i = 0; i < static_cast<unsigned int>(fishFoodModel.meshes.size()); i++)
{
    // Check when to end animation
    if ((fishFoodModel.meshes[i].mesh6DoF[tY] - elapsedTime * 0.0001f) < 0.0f)
    {
        fishFoodAnimStarted = false;
        fishFoodInit = false;
    }
    fishFoodModel.meshes[i].mesh6DoF[tY] -= elapsedTime * 0.00002f;
    fishFoodModel.meshes[i].updateModelMatrix();
    model = fishFoodModel.meshes[i].meshMatrix;
    shader.setMat4("model", model);
    fishFoodModel.meshes[i].draw(shader);
}

// Draw shark
for (unsigned int j = 0; j < static_cast<unsigned int>(sharkModel.meshes.size()); j++)
{
    // Update shark pose params
    glm::vec3 directionVec(fishFoodModel.meshes[0].mesh6DoF[tX] - sharkModel.meshes[0].mesh6DoF[t
        fishFoodModel.meshes[0].mesh6DoF[tY] - sharkModel.meshes[0].mesh6DoF[tY],
        fishFoodModel.meshes[0].mesh6DoF[tZ] - sharkModel.meshes[0].mesh6DoF[tZ]);
    float magnitude = std::sqrtf(std::powf(directionVec[0], 2.0f) + std::powf(directionVec[1], 2.
    + std::powf(directionVec[2], 2.0f));
    // Check if shark has got to food to end animation
    if (magnitude < 0.1f)
    {
        fishFoodAnimStarted = false;
        fishFoodInit = false;
        sharkModel.meshes[0].initRad = std::sqrtf(std::powf(sharkModel.meshes[0].mesh6DoF[tX], 2.
        + std::powf(sharkModel.meshes[0].mesh6DoF[tZ], 2.0f));
        break;
    }
    // Normalize to unit vec
    directionVec /= magnitude;
    sharkModel.meshes[0].mesh6DoF[tX] += directionVec[0] * 0.005f;
    sharkModel.meshes[0].mesh6DoF[tY] += directionVec[1] * 0.005f;
    sharkModel.meshes[0].mesh6DoF[tZ] += directionVec[2] * 0.005f;

    // Slowly rotate towards target
    float target = std::atan2f(directionVec[2], directionVec[0]) + float(M_PI);
    if ((target - sharkModel.meshes[0].mesh6DoF[rY]) < 0.0f)
        sharkModel.meshes[0].mesh6DoF[rY] -= 0.001f;
    else
        sharkModel.meshes[0].mesh6DoF[rY] += 0.001f;
    for (unsigned int j = 0; j < static_cast<unsigned int>(sharkModel.meshes.size()); j++)
    {
        sharkModel.meshes[j].mesh6DoF[rY] = sharkModel.meshes[0].mesh6DoF[rY] +
        0.1f * sin(elapsedTime * 5.0f + j * 5.0f);
        sharkModel.meshes[j].updateModelMatrix();

        // Set model matrix (multiply with previous model matrix for hierarchy animation)
        model = sharkModel.meshes[j].meshMatrix;
        shader.setMat4("model", model);
        sharkModel.meshes[j].draw(shader);

        // Remove "wagging" for next loop
        sharkModel.meshes[j].mesh6DoF[rY] -= 0.1f * sin(elapsedTime * 5.0f + j * 5.0f);
    }
}
```

Figure 13: Fragment shader output decided by *useTexture* boolean

The final advanced feature was collision detection which was elaborated on in the previous submission. However, this time, instead of just checking for wall collisions, I added fish tank collision detection to prevent the user from being able to walk into the fish tank. This made use of the radius of the fish tank and compared that to the magnitude of the new camera position. The boolean returned by this function (shown below in Figure **??** determines whether a valid move can be effected in the game.

```
// Position constraints check
bool checkPositionConstraints(glm::vec3 newPosition)
{
    // Only need to check XZ constraints for FPS camera
    if (newPosition.x < WALL_X_MIN ||
        newPosition.x > WALL_X_MAX ||
        newPosition.z < WALL_Z_MIN ||
        newPosition.z > WALL_Z_MAX)
        return false;

    // Check not going into fishtank
    if (std::sqrtf(std::powf(newPosition.x, 2.0f) + std::powf(newPosition.z, 2.0f)) <=
    (FISH_TANK_RAD + 0.2f))
        return false;

    return true;
}
```

Figure 14: Fragment shader output decided by *useTexture* boolean

YouTube Link

The link to my video demonstration for this deliverable is provided below. This video begins by demonstrating one of my advanced features: the shark feeding feature where the shark model changes it's route to chase and eat some pellets falling from the roof. This feature is demonstrated once more towards the end of the video. Also included in the video is the implementation of the Phong illumination model, a first person camera implementation, hierarchical kelp animations, multiple instances of textures applied to 3D models, and a variety of other animations unique to the 3D model (e.g. jellyfish bobbing, fish and shark "wagging" swimming movements, etc.).

YouTube link: https://youtu.be/9CJTs_GUVug

Declaration

I declare that all code and report writing for this submission is entirely my own work and I have not collaborated with anyone or used any code found online.

Signed: Michael Millard
Date: 16/12/2024