**Advanced Rendering and Virtual Environments**

ACW1 Report

**Underwater environment:**

**A video game screen with a blue sky and black objects

Description automatically generated**

A water surface was created using a hull and domain shader using the same principle to create the terrain and sin function to create waves. In this implementation, the functionality involves creating a procedural terrain using quad-based tessellation. The Quad structure contains tessellation factors for the edges and inside of the quad. The noise function is used to generate random values based on a given input. The grad2 function calculates gradients for noise interpolation. The fractal Noise function combines multiple noise layers to create a more complex noise pattern. The main function receives Quad input and UV coordinates, and it calculates the vertex positions for the quad based on UV interpolation. The height of the terrain is animated by adding a sinusoidal displacement to the Y-coordinate. The terrain's normals are determined by sampling the fractal noise at different positions. The final vertex positions and normal are transformed using the view and projection matrices. The result is a dynamically animated, procedurally generated terrain with realistic height variations and a natural appearance.

**Procedural sea floor*:***

A screenshot of a video game

Description automatically generated

The terrain is implemented in explicit form as a triangle mesh generated using hull shader and domain shader. The hull shader(tessellation) creates a generates heightmap-based fractal terrain using noise and fractal noise functions. The main tessellation function takes quad and barycentric coordinates as input and computes the vertex position and normal using fractal noise. The resulting geometry is passed to the pixel shader for further processing. Where I implement a fog effect at the end after material light processing.

**Vertex shader-based coral object:**

A blue square with black background

Description automatically generated

I have created a simple vertex shader that transforms a cube into a star-like object. The TransformToStar function applies two modifications to the input position. Firstly, it scales down the Y-axis to make the cube thinner, and then it applies a sinusoidal function to create the star-like effect based on the position's angle and radius. In the main function, I set the input position of the vertex to (0, 2, 0) to create a cube. Then, I apply the TransformToStar function to modify the shape into a star-like form. The final modified position is outputted in clip space, and the resulting object should as a star-like shape with altered vertex positions, creating the desired visual effect. However, there seems to be an error in my functionality.

**Pixel shader-based coral object*.***

A purple object with black background

Description automatically generated

the coral object is made implicitly as an implicit function and visualizes the object based on ray marching. The pixel shader implements a raymarching technique to render the coral object with fractal terrain and shading effects. Many functions are used to calculate distance field calculations, soft shadows, ambient occlusion, and lighting. The main function calculates the ray direction and origin based on the camera's and the current pixel's positions.

**Geometry shader-based coral object*:***

A screenshot of a video game

Description automatically generated

I have used geometry shader to create an underwater grass like coral object. The geometry shader that takes a single input point and generates a quad geometry based on the vertex position. The shader starts by checking if the vertex's y-coordinate is above a certain threshold (0.6 in this case). If the condition is met, the vertex's position is transformed by the view matrix. Then, a quad is constructed around the vertex with a specific size (quadSize) using predefined coordinates (QuadPos). Each vertex of the quad is created by offsetting the original vertex position. The quad's vertices are then transformed by the projection matrix to get their final screen position. Texture coordinates (uv) are calculated to map textures to the quad, and they are passed to the pixel shader. The output stream appends each vertex of the quad to form the final geometry, resulting in a quad that oscillates in the z-direction (using sin(time) \* 0.2) and aligns with the camera perspective as the view and projection matrices are applied.

**Reflective bubbles.**

A blue balloon in the dark

Description automatically generated

The reflective bubbles are created using ray tracing in the pixel shader. The shader uses ray tracing to render three spheres. The main function processes each pixel of a quad and simulates rays originating from the camera (eye) passing through the pixels. The rays are then intersected with the spheres to determine whether they hit any sphere surfaces. If a hit occurs, the shader calculates the shading of the surface using the Phong reflection model with diffuse and specular components. Shadows are computed by tracing additional rays from the hit point to the light source. The shader also supports reflection by recursively tracing rays from the hit point in the reflected direction. The final colour of each pixel is determined by accumulating the contributions from different depths of ray tracing. If no hit is found, the background colour is displayed. Overall, the implementation demonstrates the basic principles of ray tracing to generate realistic images of a 3D scene with spheres.

**Implicit sea plant modelling and animation:**

I am using the sin function to animate the geometry shader coral objects (plants by adding a sinusoidal movement to the z-coordinate of the vertex positions. For each vertex of the plant, the sin(time) value is multiplied by 0.2, which creates a small oscillation in the z-direction over time. This creates a waving or swaying effect for the vertices, giving the appearance of the plant animating or moving in a natural manner. The sinusoidal movement is applied to different vertices, making the entire plant appear animated and dynamic when rendered on the screen.

This same concept is implied in other shaders as well.

**Novelty and own effects:**

A computer keyboard with a blue sky

Description automatically generated  
For novelty effects, I have added a fog effect to terrain, water and sea plants as well as an underwater tint effect.

**Fog:**

The fog effect is implemented by calculating a fog factor based on the distance from the camera (view position) to the pixel being rendered. The fog factor is used to interpolate between the original pixel colour and the fog colour, creating a gradual blending effect that simulates the fog.

**Underwater Tint:**

First, transform the pixel position from Normalized Device Coordinates (NDC) to clip space and then calculate the view direction in world space by applying the inverse view-projection matrix. The distortion strength and refraction index parameters are used to control the intensity of the effect. The distortion offset is computed based on the view direction and distortion strength. The original pixel colour is sampled before applying the underwater effect. The effect combines an underwater colour tint with the original colour, creating a tinted version. The texture is then sampled with distortion applied to simulate refraction. The final pixel colour is obtained by blending the underwater tinted colour and the distorted colour using the refraction index. This process results in a final pixel colour with the combined underwater tint and refraction, producing a visually convincing underwater effect.