# Tropical Coral Reef

Proteeti Kushari
Student Id: 24333831
Computer Graphics (CS7GV6), 2024-25

## Introduction

This project aims to use OpenGL in C++ to simulate a vibrant underwater system with multiple schools of fish and a coral-enriched sea-bed. It also serves as a programming exercise allowing us to understand and use the very basic principles of a computer graphics pipeline so as to create a coherent and interactive scene without the aid of any specialized tools.

A demo of this project in action can be seen at: `https://youtu.be/J_IxNGR8M5s`

## Key Libraries and Concepts

### Model Loading and Extraction

The aiImport method from the Assimp library is used to load pre-downloaded 3D models into the project. This method is called with a variety of available flags so as to ensure the smoothest reading of the model into a local mesh structure.

| Flag | Purpose |
|---|---|
| aiProcess_Triangulate | Converts all model faces into triangles |
| aiProcess_OptimizeMeshes | Merges redundant meshes into a single mesh and reduces the number of draw calls |
| aiProcess_GenSmoothNormals | Generates smooth normals for each vertex by averaging the face normals of adjacent polygons |
| aiProcess_ImproveCacheLocality | Reorders mesh vertices so as to improve cache performance |
| aiProcess_CalcTangentSpace | Calculates tangents and bitangents for each mesh vertex |
| aiProcess_PreTransformVertices | Applies all model transformations to the vertices such that the resulting model doesn't have any hierarchies |

Table 1: aiImport Flags

Once the model is read successfully, the converted mesh is mapped into the readable local ModelData format by extracting the vertices, normal, and texture coordinates of the same. This version is then passed on to the rest of the code for rendering.

## Lighting and Shading

- **Light Source**: The light is pointed at the centre of the view-port coming from the top (height mapped by the y-axis). When viewed from the side (main view), the light diffuses nearly equally.

- **Shading Model**: The primary shading scheme used in this project is a basic implementation of the Phong shading model. In this model, ambient, diffuse and specular light components are used to to simulate how falling light bounces off of each model [5].
  For the fishes and plants, I have used a single-colour shading format ensuring they remain clearly visible even under the water overlay. For the terrain, I used a Phong shading model in conjunction with a simple algorithm to generate a variety of colours for the reef base. This algorithm was applied to the fragments of the terrain to ensure fast rendering:

$$\begin{aligned} \text{colour}_{\text{red}} &= |\sin(\text{position}_x)| \\ \text{colour}_{\text{green}} &= \left|\sin\left(\text{position}_y\right)\right| \\ \text{colour}_{\text{blue}} &= |\sin(\text{position}_z)| \end{aligned}$$

  Where:

  position : position of the fragment in the 3D viewport with $x, y,$ and $z$ components

## Texture Mapping

Texture mapping in projects dealing with Computer Graphics allows us to give the project components a more realistic look. In this project, I have used the stb_image library to read and parse the texture images [7].
After parsing, the image is uploaded to the GPU, and mipmaps are generated. Mipmaps help speed up the rendering process when we observe the model from different distances.
Once the texture has been loaded into the system, a vertex buffer object (VBO) and a vertex array object (VAO) are used to map the texture coordinates smoothly over the entire screen.

## Rendering Pipeline

The graphics pipeline for rendering the models is divided into several stages: vertex processing, transformation, and rasterization. We begin by loading the model's vertex and normal positions, and storing them in buffers. Two Vertex Buffer Objects (VBOs) are created for the positions and normal respectively, which are then linked to the shader attributes. A Vertex Array Object (VAO) is also added to preserve the binding state of these attributes and buffers. The vertex shader program then processes the input vertex data to transform the vertices from the local object space to the clip space. The model matrix defines the model's local transformations. The view matrix captures the camera's perspective. The projection matrix maps the 3D scene onto the 2D screen using a perspective projection. These matrices are then linked to the shader through uniform variables and transformations are applied across all vertices.

Most models used in this project also support hierarchical animations, with 'child' models being defined and linked to the main models, ensuring geometric and aesthetically pleasing motion patterns.

Finally, for rendering on screen, the transformed vertices of the models are rasterized into fragments by the OpenGL pipeline. Each fragment signifies one pixel on the screen, and its positions and normals are calculated from the vertex data. A fragment shader then processes these fragments, applying lighting calculations and colour to simulate the appearance of the model. The `glDrawArrays` function in the end is used to draw the models based on the triangular primitives that are mapped by the vertex data.

### Procedural Terrain

The base terrain is generated using the Perlin noise algorithm [6]. We begin by generating a heightmap for our terrain, using the Perlin noise simulator from the stb_perlin library. a scaling factor of 0.2 is then used to generate semi-sharp ridges. We then generate the terrain mesh by mapping the vertices and indices of the same. Then these individual vertices are structured to form triangles, enabling OpenGL to successfully parse and render it. We then apply a rough-coral texture on it using an image imported into the project [1] using the stb_image library.

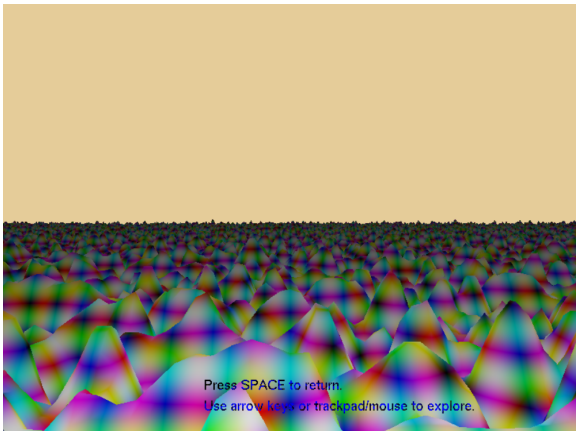


Figure 1: 3D terrain rendered with Perlin noise

## Implementation

This project has been implemented entirely using C++ and public OpenGL Libraries such as Assimp, stb_image and stb_perlin. The code was developed in Visual Studio Community 2022.

## Modelling and Scene Building

The scene is built with a number of individual structures interacting with each other. The components consist of pieces that are both imported from third party sources and generated as part

of the developed code. In this section we discuss all components except for the procedural terrain which has already been covered in the previous section.

## Fish

Two schools of fish, imported as 3D models [4], are animated using a sinusoidal animation algorithm and they traverse the visible screen crossing one another and weaving through the plant simulations. Each school is composed of 11 fishes. The bigger fish is the main model which is animated alongside 10 iterations of the same mesh reproduced as evenly spaced child models. The motion algorithm controlling the sinusoidal movement of the fish is further discussed in a subsequent section.

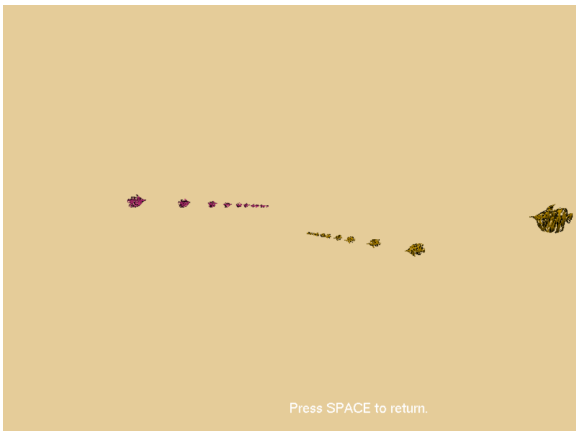


Figure 2: Fish schools rendered in empty screen

## Plants

- **Dynamic Green Shrub**: This animated shrub is made of a 3D model [3] which is then animated using child models which are reproductions of the same parent mesh. The motion is simulated through a controlled, yet randomized, variance in size and position over 10 iterations.
- **Red Coral Plant**: This red shrub, composed of the same 3D model as the green plant, is reciprocated twenty times using a semi-randomized algorithm to define its size and position in the scene. This aids in the vibrancy of the scene.


Figure 3: Red and green plants rendered in empty screen

## Water Overlay

A simple texture overlay is imported [2] into the GPU and used to simulate the water texture, covering the terrain, through efficient mipmap generation. The waves are simulated in the fragment shader, using the system time for incrementing the simple amplified sine and cos curves. The below equations show the formula that is used to generate these waves.

$$y = 1.26 \times \sin(10x + \text{time})$$

$$x = 1.26 \times \cos(10y + \text{time})$$

Where:
x: current x-axis position of the texture coordinate y: current y-axis position of the texture coordinate time: current time

## Motion Control

### Keyboard Input

The arrow key presses are recorded and used to shift the camera view intuitively. A slight rotation component is also added to the up and down arrow keys, exploiting the 3D nature of the scene.

### Trackpad/Mouse Input

The mouse/trackpad marker position is tracked constantly in such a way that the user is able to click-and-drag to navigate the scene in a 3D frame.

### Automatic Motion

This motion algorithm is added primarily for controlling the motions of the two schools of fish.

$$\Delta = (\text{current time} - \text{previous time}) \times 0.001$$

$$x = (x - 2\Delta) \mod 50$$
$$y = y - \Delta + \sin(10y + \Delta) + \cos(3y^2 + \Delta)$$

Where:
x: current x-axis position of the fish model and child
y: current y-axis position of the fish model and child

We further use selected negative values to translate the model cluster of the second (pink) school of fish, so as to reverse its direction, thus simulating the two schools weaving through each other midway through the viewport.

## Camera Control

### Top View

This is the initial view when the project is launched. It shows all the models and other components from a camera view positioned high along the y-axis. For the sake of performance, the motion control functions are not enabled in this view. The viewer is prompted and able to press the SPACE key to toggle to the main view and back.

### Main View

This is the primary view for observing the dynamics of the underwater scene. Here, the camera is positioned along the positive z-axis. Here too, the user is allowed to press the SPACE key to return to the initial, top-view.

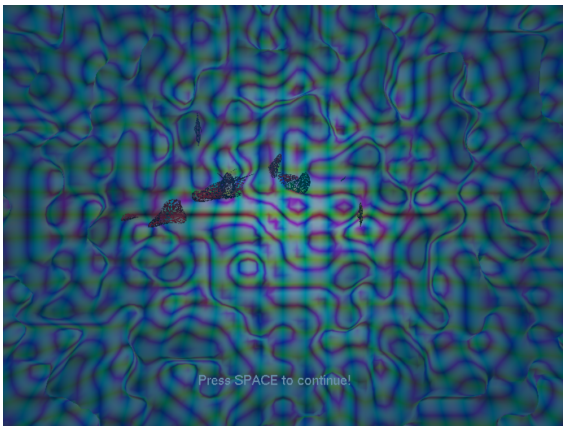


Figure 4: Top view of 3D terrain and models

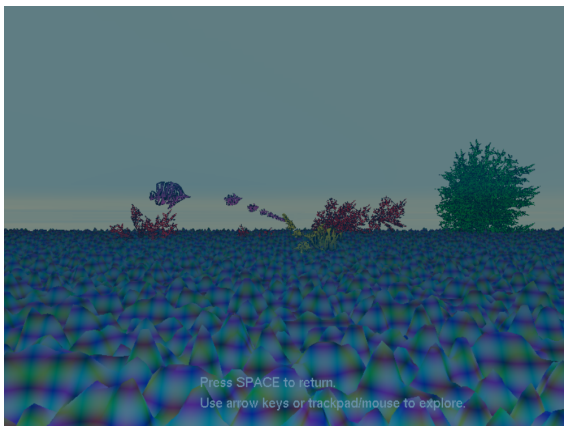


Figure 5: Main view of 3D terrain and models

## Text Prompts

Appropriate text prompts are provided for the user to follow for all possible input actions i.e. the toggling of camera view via pressing the SPACE key and navigation using the arrow keys.

# Limitations and Future Improvements

This project is limited in its implementation complexity owing to the development device limitations. However, we must consider the avenues to further improve this vibrant scene.

## Plants Scattering

The scattering algorithm for the red plant can be improved to achieve a greater randomized effect. This will also be enhanced by increasing the number of instances, which in my case I have not implemented due to limited processing capacity.

## Fish Animation

The animation can be made faster and smoother by decreasing the delta time used to compute the sinusoidal wave. A marginal improvement can also be observed upon running the project on a higher capacity computer system.

# Conclusion

This model is a relatively simple recreation of an incredibly complex and diverse natural scene. I have used the basic principles of Computer Graphics to simulate an underwater reef which has allowed me to understand the underlying steps of a graphics pipeline in a modern computer system.

It involves the exploration of special features such as hierarchical animation, procedural terrain generation, user-triggered view-port manipulation, multiple camera angles and animated water overlay.

# References


1. Freepik, `https://www.freepik.com/free-photos-vectors/coral-reef-texture`
2. CGTrader, `https://www.cgtrader.com/items/3206924/download-page`
3. Free3D, `https://free3d.com/3d-model/coral-v1--901825.html`
4. TurboSquid, `https://www.turbosquid.com/es/3d-models/fish-yellow-3d-model-1482528`
5. Anton Gerladen, `https://antongerdelan.net/opengl/shaders.html`
6. Omar Delarosa, `https://www.omardelarosa.com/posts/2020/01/06/infinity-terrain-in-c++-using-perlin-noise-and-opengl`
7. Learn OpenGL, `https://learnopengl.com/Getting-started/Textures`