# COMP 371: Computer Graphics
# Final Project Documentation

Mohamad Ghanmeh 40062339
Nicholas Joannette 27439059
Adrien Kamran 40095393
Sami Merhi 40136648
Ale Niang 29412360

August 22, 2020

# 1 Description of the Project

The goal of the project was to enhance our understanding of computer graphics principles and the design patterns involved in developing world rendering procedures with Modern OpenGL and C++. Our linear algebra and algorithm development skills were tested during this project, as we implemented the collision detection, mesh generation, chunking, free quaternion rotation, spherical linear interpolation, and animation systems which would allow us movement and interaction with an infinitely generated procedural world. As we researched mesh generation and manipulation, we were able to implement the 'Cube Marching'. This marching cube algorithm enabled us to generate an infinite and procedural world from the vertices of cubes, complete with terrain collision detection and the placement of models flush on surface faces - alongside many other features described in the following documentation. This result of this project is a combination of techniques learned and applied in the first two assignments with new techniques that were researched, implemented, and innovated to generate our cavernous underwater world.

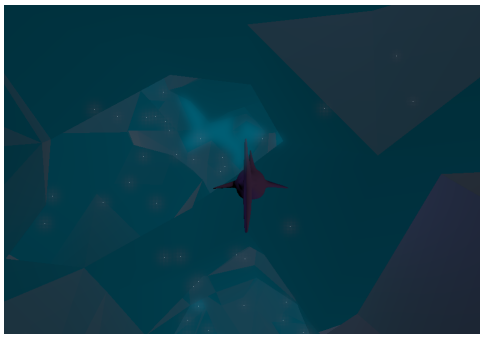

Figure 1: Caverns

Our team chose option 2: 'Walk Through a Procedurally Modeled Pixelated World', as we had been intrigued by the possibility of generating a world out of cubes by using cube marching. We wanted to produce a world in which we would be able to move around using an animated 3D model as though it were a third person experience. We saw this project as the opportunity to make an ambitious attempt at an infinitely large procedurally generated world in an underwater cavern system. The following report will demonstrate our objective

when doing this project, the reason in which each teammate was interested in this concept of the project, how we accomplished the object, what we learned throughout this experience and the lists of references and resources we used to produce the final results of our project.

## 2 Why This Project Interested Us

We chose to create a procedurally generated world as we had been inspired by our research on the 'Cube Marching' - also known as 3D surface reconstruction - algorithm pioneered and demonstrated by Lorensen and Cline in 1987, and expanded upon by Paul Bourke in his article 'Polygonising a Scalar Field'. We were also motivated by the concept of an infinitely large procedural world generated by using a 3D chunking system - much like Minecraft or No Man's Sky.

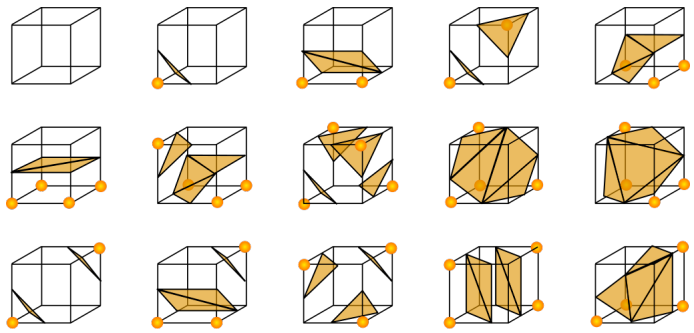

Figure 2: Marching Cubes Surfaces

As we toyed around with the cubic 'Cube Marching' mesh generation algorithm, we noticed that the structures began to look much like underwater caves - and were excited about the idea of moving around the world, controlling an animated shark or whale as our method of navigating throughout the world we generated. We wanted to be capable of implementing multiple light sources - and thought it would be interesting to incorporate glowing crystals using an HDR exposure and blur effect. We believed we could create a realistic underwater effect by combining the illusion of underwater fog - as the sea is not as bright as the water we see in pools or any filtered water. We thought it would be interesting to use a soundtrack like that of Subnautica, and small 'thudding' sounds upon collision to

fit our underwater experience. We had many more ambitions such as procedural grass and other NPC species which roam the world, but which we did not have time to implement. The following sections will demonstrate how we tried to accomplish our goals using the best method we could create.

# 3 How we accomplished our project

## 3.1 The PA1_Practice.cpp file description

A header file (Shader.h) was created to handle and modularize the process of instantiating shaders such has ".vs/.fs/.gs" files. The vertex and fragment shaders are compiled into individual programs, with transformation matrices and lighting attributes included by default. The same goes for materials, which take lighting attributes (ambient, specular, and diffuse RGB values) in the constructor.

Because we were implementing secondary framebuffers to incorporate the HDR exposure and bloom effects, we implemented the renderQuad() function as gleamed from LearnOpenGL, to implement high dynamic range lighting and bloom/blur effects by manually rendering our frame to the screen. The textures attached to the buffers are updated on window resize such that the dimensions remain consistent. This is used later in with the "blurShader.h" file to create a gaussian bloom spread effect.

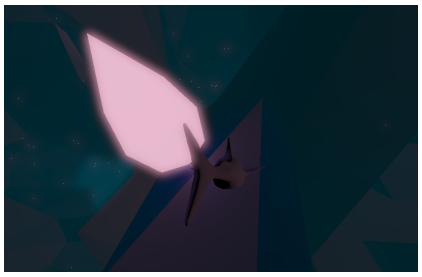

Figure 3: Bloom

Depending on hardware, your frame rate can differ when running an OpenGL program. As such, we felt that it was necessary to create a method which could normalize the "speed" values for movement relative to the framerate. In turn, this method eliminated cases where the camera would move extremely quickly on faster machines, and vice versa

Our project has a controllable player model, as well as an environment filled with obstacles. It was necessary to implement a collision system that would allow the model (oriented using quaternions) to bounce off of the faces of generated terrain meshes in a smooth and realistic fashion. a collision is handled by testing if a surface has entered contact with an invisible sphere around the model. A "thud" sound is played for feedback. The model's front vector is reflected on the surface while maintaining its initial speed, using a quaternion rotation according to the angle of incidence.

## 3.2 The SkinnedMesh.h file description

For our project to support imported 3D models, we used the Assimp library in combination with guidance from ogldev.atspace.co.uk to implement animated and non-animated modelS. Model data is read, stored, and processed for use in the GPU iteratively through the vertices and recursively through the nodes. Afterward, the diffuse/specular/normal/height maps are loaded texture by texture, and materials are also supported.

[PLAYER.H] The Player's model is controlled, rendered, and interacted with through the code in the Player.h file, Variables related to the Player's orientation in the world, the Player's acceleration, and velocity, and the Player's linked model are stored here (among others). The player model

## 3.3 The UI_InputManager.h file description

This is by far the most complex and interconnected code we have, next to the main file. Everything (models, camera, shaders, lighting/shadows, ImGui, etc.) is actively controlled or can be controlled through the inputs registered in this file.

A set of shaders was required to render the marching cubes algorithm visually (see FIG. 4). In FIG. 10-1 and 10-2, standard inclusions for a fragment shader are present, such as uniforms for the textures and colors, but new attributes have been created specifically for unique visual effects. A gradient fog uses the depth of the scene to filter the light and change its color, and this attenuation affects the base color present on the geometry. In FIG. 10-3, the vertex shader uses standard uniforms, and a normalization function is included in the

main to account for the use of normals in the main file.

### 3.4 Blender usage

Blender was used briefly to adjust the sizing of the animated shark model, as well as the submarine and crystal models.

## 4 What we learned in this project

Much import information was learned during this project, including but not limited to:

- Dynamically generating meshes based on potentially random input using Cube Marching
- Handle collision detection with procedurally generated polygon faces
- Developing a 3D chunking system for infinite procedural generation
- Implementing a buffer pooling design pattern and more complex recycling of VAOs and VBOs
- Incorporating animated models through recursive node hierarchies
- Implementing multiple light sources
- Using FrameBufferObjects to create cool visual effects

While we didn't accomplish all that we set out to acheive - eg. point-light cubic shadow maps which incorporate multiple light sources, or procedural grass, this was an intensive project and we are only saddened that we could not spend more time to further develop it - as we will after the course!

# 5 Figures index