Eötvös Loránd University

Faculty of Informatics

Department of Media and Educational Technology

Procedural Terrain and Volumetric Cloud Generation for Interactive Gameplay

Supervisor:
Szabó Dávid
Assistant Lecturer

Author:
Rajinish Aneel Bhatia
Computer Science BSc

Budapest, 2025

Contents

Chapter 1

Introduction

This thesis explores the implementation of real-time procedural terrain generation and volumetric cloud rendering using C++ and OpenGL. The goal is to create a fully explorable 3D environment in which the user can pilot an aircraft through dynamically generated landscapes and clouds, all while maintaining high rendering performance.

Procedural content generation enables infinite, non-repeating environments without the need for handcrafted assets. Volumetric clouds further enhance immersion by simulating complex atmospheric phenomena. Together, these systems form the foundation of a lightweight, yet visually rich virtual world.

The main inspirations for this work come from Sebastian Lague's development series [1] and community shaders such as those found on Shadertoy [2] [3]. However, unlike these examples — which either depend on high-level engines or focus on isolated effects — this project aims to build everything from scratch, including the rendering pipeline, asset management, and interaction systems.

Following conventions are used throughout the document:

- **Vector:** $\vec{x} = \langle a, b, ... \rangle$
- **Unit Vector:** $\hat{x}$
- **Length of a vector**: $\|\vec{x}\|$
- **A point**: $\mathbf{x}$
- **Matrix**: A
- **Dot product of two vectors**: $\vec{a} \cdot \vec{b}$ or $\langle \vec{a}, \vec{b} \rangle$
- **Cross product of two vectors**: $\vec{a} \times \vec{b}$

Chapter 2

User Documentation

To ensure optimal performance consistent with testing, the recommended hardware includes an Nvidia GeForce 3050 Ti (with a minimum of 4GB VRAM) and at least 8GB of system RAM. The game is distributed as a ZIP archive specifically for Windows; all necessary files are included, and the game can be launched directly by running the executable. For other operating systems, the game must be built from the provided source files.

Windows does not automatically use the Nvidia graphics card when you run an OpenGL application. You need to manually configure this by navigating to System > Display > Graphics then browse for the game and choose high performance in graphics preference.

2.1 The goal of the game

This project serves mainly as a demonstration of the implementation so no complicated features are implemented. However, to make the game playable the user is assigned with a single task. The goal of the game is drop packages at some predefined coordinates which are randomly selected as soon as the package is dropped. The package can be dropped in some predefined circle of radius around the target coordinates. So the steps can be summarized as:

- Fly around and explore the world.
- Fly to the target coordinates.

- When you see a sign on the screen, drop the package. And go to the next coordinates.
- Settings can be manually configured as desired, which change the shape and scale of the clouds.
- The airplane's properties can also be configured using the menu.

A navigation bar is also added at the bottom of the screen which is just a projection of the players coordinates on the x-z plane, this bar can be used for reaching the target coordinates as will be momentarily described.

2.2 Playing the game

Figure 2.1: First look

The image above provides a first look at the game, the config menu can be hidden using either the cross button or by pressing the key **F1**. Currently there is no way to change the airplane model from inside the game, the only way to do that is to change the source files.

2.2.1 The controls

Following keys can be used to control the aircraft:

W: Go forward (apply thrust)

D or →: Tilt the plane towards right.

A or ←: Tilt the plane towards left.

↑: Tilt the plane downwards.

↓: Tilt the plane upwards.

space-bar: Drop the package

Q: Rotate the aircraft around the UP axis to the left.

E: Rotate the aircraft around the UP axis to the right.

2.2.2 Understanding the navigation bar

The navigation bar can be understood by looking at the picture below

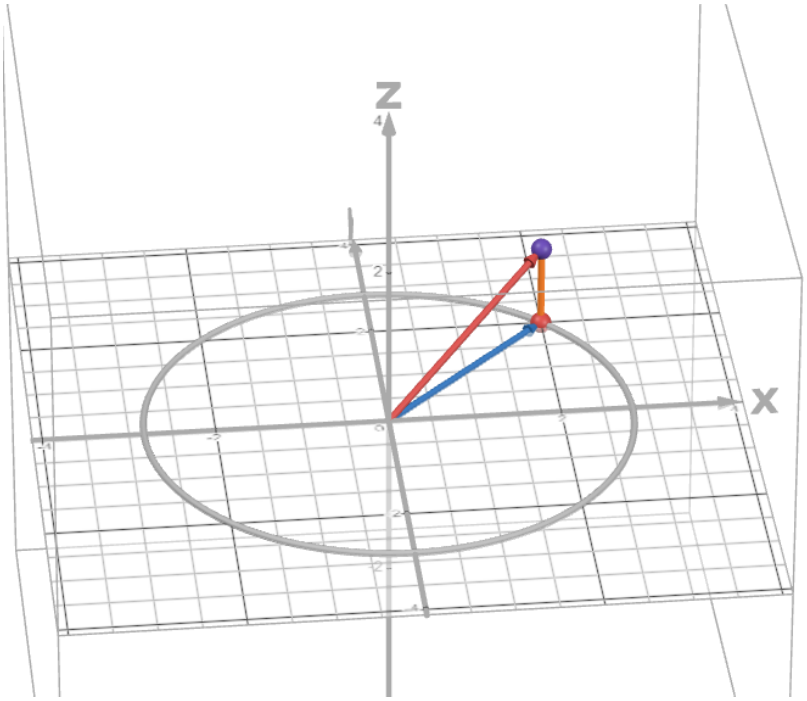


Figure 2.2: Projection of a point in 3d onto the plane

If the vector in red is current velocity vector of the airplane and the blue vector is its projection onto the horizontal plane then this vector is exactly where the black arrow points in the navigation bar.

The blue blinking dot is where the current target position is, it is relative meaning that the distance is scaled and capped so it remains within the navigation bar but if you're closer to the target then the blue point moves inside the circle indicating that you're near the target.

2.2.3 Dropping the package

When you are within the pre-specified acceptable radius of the target a text as show in Figure 2.3 appears on the screen.

At which point you can press **space-bar** to drop the package and move to the next assigned point in the navigation bar.

Figure 2.3: Text indicating the drop point

2.3 Understanding the config menu

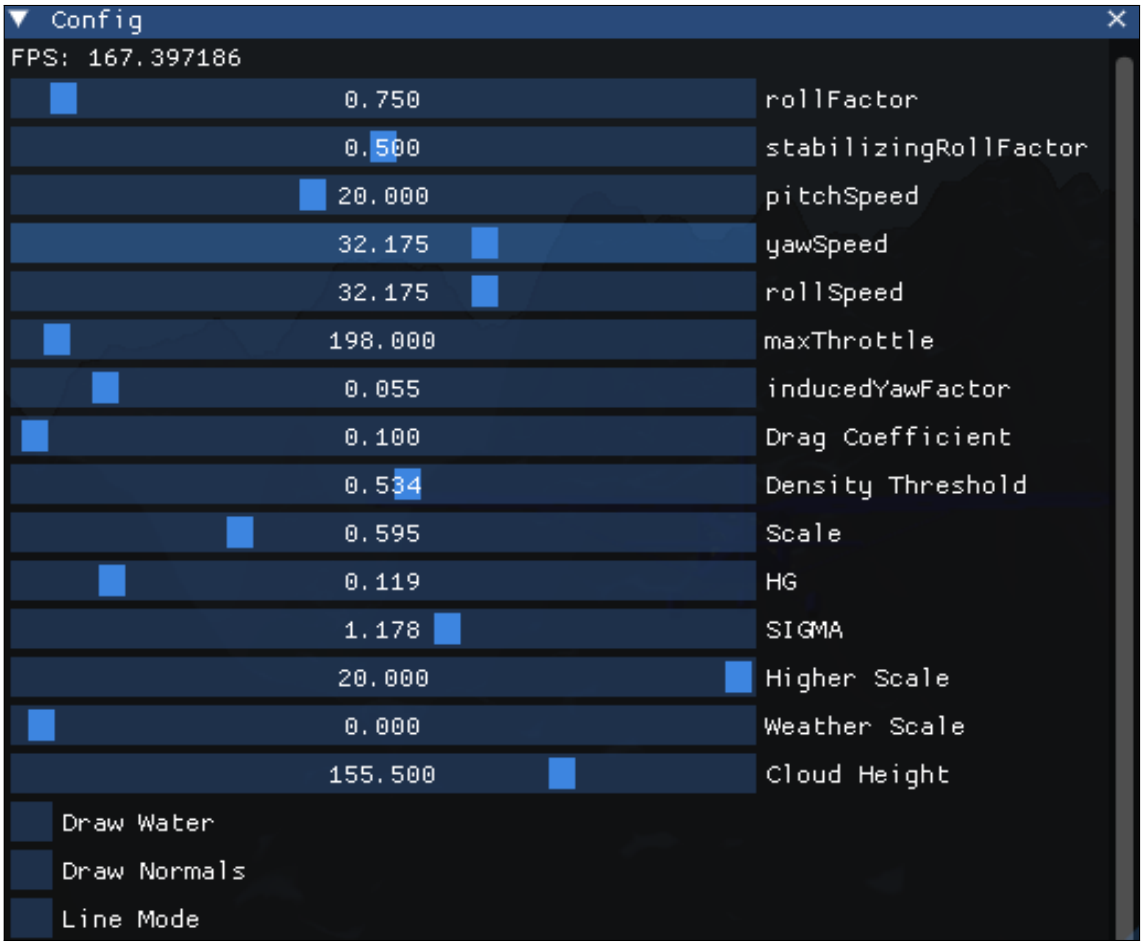


Figure 2.4: The configuration menu

The config menu was built using ImGUI library [4]. The first 7 configuration parameters (until **inducedYawFactor**) are for the aircraft and they control/define the following things:

Aircraft parameters	
Parameter	*Description*
rollFactor	How much emphasis should be put on the roll control given by the user, the total roll torque is a weighted sum of user's roll and the stabilizing roll.
stabilizingRoll	How much emphasis should be put on the stabilizing the aircraft, for example, if the aircraft is traveling fast it has a tendency to align itself towards the center, this parameter controls how strongly that should happen.
pitchSpeed	Factor by which changes to the current pitch should be applied.
yawSpeed	Factor by which changes to the current yaw should be applied.
rollSpeed	Factor by which changes to the current roll should be applied.
maxThrottle	Factor by which the force in the forward direction should be applied when the user presses **W**
inducedYawFactor	When the airplane tilts around the z axis, i.e it rolls there is also an induced yaw on it, this parameter determines how strong that induced yaw should be.

Table 2.1: Aircraft parameters

Weather parameters	
Parameter	*Description*
Density Threshold	The clouds have density between 0 and 1 at points in space, this determines the minimum density the point must to have to be rendered.
Scale	The scale at which the clouds should we sampled from the noise textures, lower values result in bigger fluffier clouds, higher values make them smaller and closer.
HG	The Henyey-Greenstein constant (further described in developer documentation).
SIGMA	The transmittance factor of the clouds.
Higher Scale	The scale at which noise from high frequency noise textures should be sampled from. Empirically, this was found to give good results between 15 and 20.
Weather Scale	The scale at which the parameters for the weather (i.e where clouds can be) should be sampled from the weather texture (due to the noise textures I have, this value should be extremely small).
Cloud Height	Height at which the clouds should be.

Table 2.2: Weather parameters

Terrain Parameters	
Parameter	*Description*
Draw Water	Draw water over the terrain, the default height is 5.0 (not alterable).
Draw Normals	Draw normals for the terrain, the default color is white (not alterable).
Line Mode	Draw the terrain in line mode (just the edges of the triangles), to see the effects of LOD.

Table 2.3: Terrain Parameters

Chapter 3

Developer documentation

3.1 Building the code

There are currently two ways of building the code on Windows OS either with Visual Studio and using **vcpkg** as the package manager or with **MinGW** and manually building the packages (which I have deprecated but it's possible and I have put it in a separate branch).

First the following steps must be fulfilled to start developing (on windows):

- Install **vcpkg** and add it to your path.
- Run the integrate install command so Visual Studio can detect it.
- Install CMake.

3.1.1 Building with CMake

To make it easier to download the required packages, a powershell script `install_dependencies.ps1` is provided with the code which installs all the required dependencies.

Then the project can be built with CMake either by running `build_for_vs.ps1` script or by doing the following in the shell:

```
1 mkdir build
2 cd build
3 cmake .. -G "Visual Studio 17 2022" -A x64 -DCMAKE_TOOLCHAIN_FILE=%
    PATH_TO_VCPKG%/scripts/buildsystems/vcpkg.cmake
```

Code 3.1: Building with CMake

You can replace generator with your compiler of liking, MinGW, for instance (if you have the packages installed). Replace the tool chain file path to your vcpkg path.

CMake was chosen because of its cross-platform support. If, for instance, you want to build on linux and have the required packages then you can build using the same CMake file, in which case it will generate a Makefile instead of the VS solution.

3.2 Design and the rendering pipeline

This sections describes how the components fit together and in what order the things are rendered before proceeding to explain all steps in detail.

3.2.1 The OpenGL Rendering Pipeline

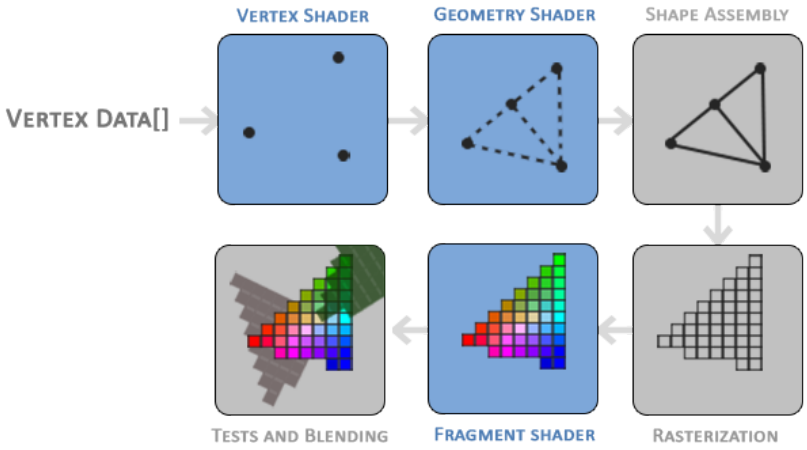


Figure 3.1: The OpenGL Pipeline
[5]

First let's look at the OpenGL rendering pipeline in Figure 3.1. The only procedures that we are concerned with at the moment are Vertex and Fragment shaders. Geometry shaders will be described at a later stage when they are needed.

Definition 1 (Shaders)**.** Shaders are programs that run on the GPU, there are various ways to send data to shaders. In the code, I mainly send data through vertex buffers or through uniforms.

Without going into too much depth, vertex shaders transform vertices from local space to normalized device coordinates and fragment shaders are used for choosing the colors (as depicted in the picture above).

3.2.2 Vertex Shader Convention

The code snippet below describes how a typical vertex shader looks like in the code:

```
layout (location = 0) in vec3 aPos;

uniform mat4 proj;
uniform mat4 view;
uniform mat4 model;

void main() {
  gl_Position = proj * view * model * vec4(aPos, 1.0);
}
```

Code 3.2: Vertex shader convention

In this document, I will be representing the projection matrix as $\mathbf{P}$, view as $\mathbf{V}$, and model as $\mathbf{M}$. In some vertex shaders there is also a local transformation matrix because I sometimes prefer to dissect the model matrix into two matrices the model and the local matrix, the former puts the object into the world space and the local transformation is responsible for any rotation or scaling.

3.2.3 Shader loader

To make it easier to load, compile and use shaders the engine comes with a Shader loading class. The design of the shader loader/manager is inspired by the one on LearnOpenGL [5]. This class is responsible for allocating and destructing the shaders. A different class is also defined specifically for loading Compute shaders which will be described later. However, the interface for using and setting the uniform variables is identical for both classes.

Uniform variables can easily be set using this shader class, as illustrated in the code snippet below (all such functions can be found in the header file):

```
Shader myShader {"vertexSource.glsl", "fragmentSource.glsl"};
myShader.use();
```

```
3 myShader.setVec3("cameraPos", glm::vec3(0.0f));
```

Code 3.3: Shader class usage example

3.2.4 Mesh handling

In OpenGL the mesh data must be transferred to a **VBO (Vertex Buffer Object)** first and the **VBO** must be bound to a **VAO (Vertex Array Object)**, instructions for how to parse the data in **VBO** must also be explicitly defined and optionally a drawing order of vertices can also be defined in a **EBO (Element Buffer Object)**. To make this simpler a **Mesh** class comes with the engine. This is also inspired by an implementation on LearnOpenGL [5]. The figure below gives a rough UML of how the mesh class is structured.

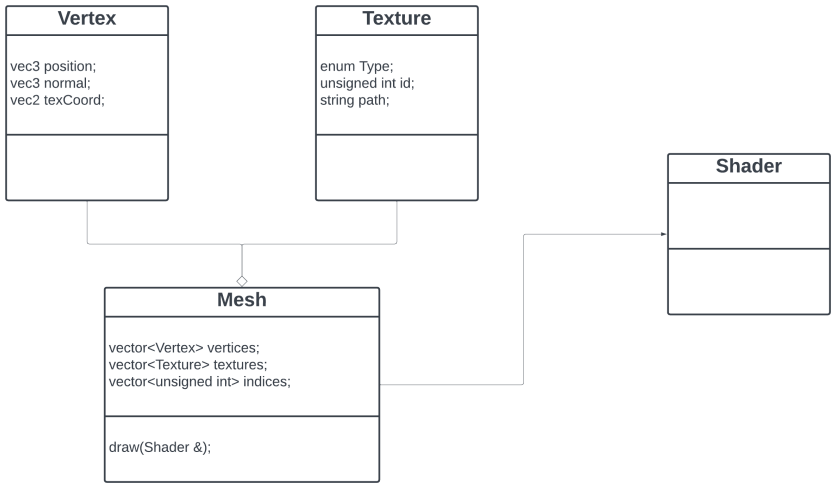


Figure 3.2: Mesh Class
[5]

Meshes for 3d can be generated easily using a mapping $f(u, v) \to \langle x, y, z \rangle$. Some simple examples are given in `funcs.h` and `funcs.cpp`. For instance, a sphere can be generated using the function: $f(\theta, \phi) \to \langle \cos\theta \cos\phi, \sin\phi, \sin\theta \cos\phi \rangle$, where $\theta \in [0, 2\pi], \phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. The normal vector for a sphere $\hat{n}$ is, of course, the same as $f(\theta, \phi)$. An implementation for generating the mesh of a torus is also given; a simple version of which can be derived by taking the parametric equation of a circle and translating it along, say, the x axis by $r\prime$: $C(\theta) = \langle r\cos\theta + r\prime, r\sin\theta, 0 \rangle$. If $\mathbf{R_y}(\phi)$ is the rotation around y axis then the torus is $f(\theta, \phi) = \mathbf{R_y}(\phi)C(\theta)$ where $\theta, \phi \in [0, 2\pi]$.

3.2.5 Model loading

The first implementation was done with a custom object file loader, however, it is infeasible to write a complete model loader that triangulates the vertices, supports multiple formats, and pares the texture files accurately. So, the decision to use **assimp** was made. The **Model** class is basically a wrapper around **assimp** functionality. This implementation was also motivated by the one provided on LearnOpenGL [5], however, contrary to that implementation this one is more optimized and handles the loading of materials more accurately.

The figure below can be used as a reference for understanding the code in the Model class.

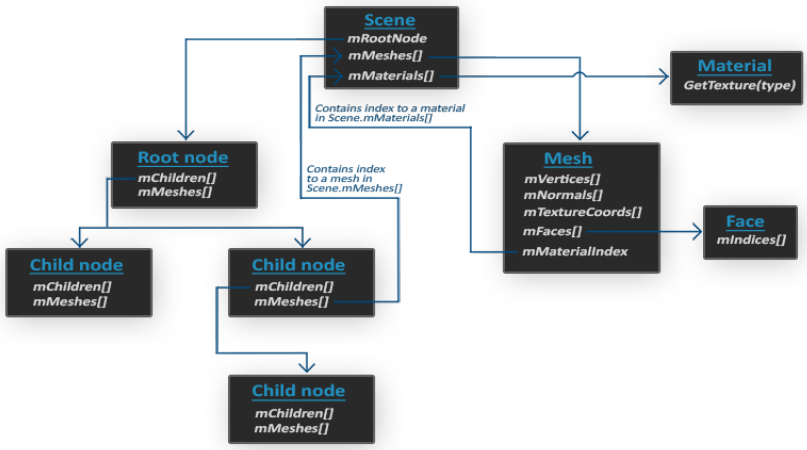


Figure 3.3: Assimp Structure
[5]

A model can be thought of as a collection of meshes, so the rough UML diagram of the class below should make sense:

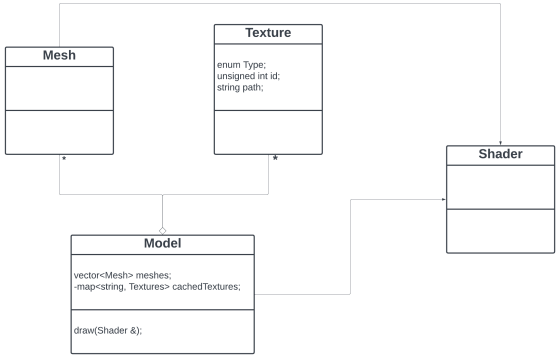


Figure 3.4: Model class UML

Since models can have multiple diffuse and specular textures, they must be defined in the shaders in following convention. Diffuse textures must follow the following naming **texture_diffuse[1..]**, specular textures must be named as **texture_specular[1..]**.

3.2.6 `Camera` class and the camera transform

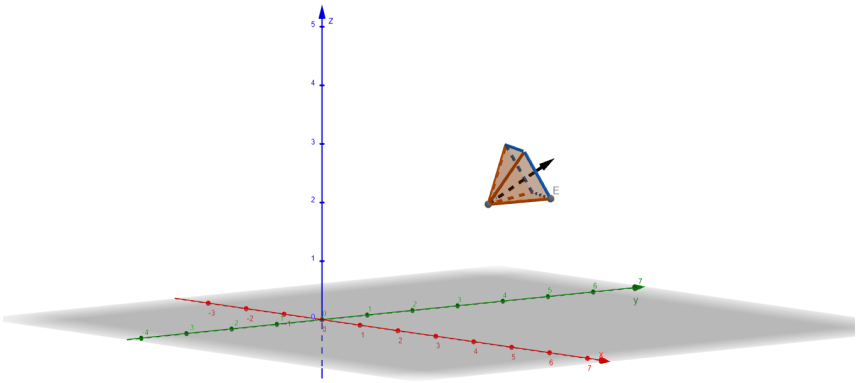


Figure 3.5: Simple Camera example

The camera can be imagined as a vector in the figure above. It sits at a point $\mathbf{p}$ looking in direction $\hat{\mathbf{d}}$. Contrary to some common implementations, which store a lookAt variable to store what point the camera is looking at, I only store the direction which can be dissected into two components: pitch (rotation around the $x-axis$), represented as ϕ, and yaw (rotation around the $y-axis$) which is represented as θ. Similar to what was done in the meshes section 3.2.4, it can be observed that these define a spherical coordinate system and we can get $\hat{\mathbf{d}}$ as follows:

$$\hat{\mathbf{d}} = \langle \cos\theta \cos\phi, \sin\phi, \sin\theta \cos\phi \rangle \tag{3.1}$$

However, the pitch is constrained to avoid distortions so, $\phi \in [-\frac{\pi}{4}, \frac{\pi}{4}]$. An absolute up direction vector is also defined for the camera which I will denote as $\hat{u}$ and $\hat{u} = \langle 0, 1, 0 \rangle$. The projection matrix works by assuming that the camera is looking in the negative $z-axis$ and is sitting at the origin. So $\mathbf{V}$ must be the matrix that puts the camera in this position. $\mathbf{V}$ must first translate the camera to the origin and then apply the inverse rotation of the current camera rotation.

If $\vec{p}$ is the position vector of the camera, then let $\mathbf{T}_{-\vec{p}}$ represent the translation matrix that translates by $-\vec{p}$.

The current rotation matrix of the camera can be obtained by getting the front (but flipped because the camera starts by looking in the negative z-axis), right, and up vectors of the camera:

$$\hat{f} = \text{front} = -\hat{d}$$
$$\hat{r} = \text{right} = \hat{d} \times \hat{u}$$
$$\hat{a} = \text{up} = \hat{r} \times \hat{d}$$

The rotation matrix is then:

$$\mathbf{R} = \begin{bmatrix} \hat{r} & \hat{a} & \hat{f} \end{bmatrix}$$

This can be imagined as where the $\hat{i}, \hat{j}, \hat{k}$ vectors land after the transformation.

We want the inverse of this matrix. Since this is an orthonormal matrix, the inverse is the transpose of the matrix:

$$\mathbf{R^{-1}} = \mathbf{R}^T$$

Thus, the final view matrix is:

$$\mathbf{V} = \mathbf{R}^T \mathbf{T}_{-\vec{p}}$$

The `Camera` class in the code provides the above functionalities, and the view matrix can be easily obtained by calling the `camera.getView()` method.

3.2.7 Audio Manager

The engine includes an audio manager capable of playing 2D sounds. Currently, there is no implementation available for 3D audio playback. This component serves as a wrapper around the functionality provided by **OpenAL**. Audio files are loaded using **libsndfile**, and the engine currently supports only the **WAV** format.

The following code snippet demonstrates a basic usage example:

```
#include <audio_manager.h>
#include <iostream>

int main()
{
    AudioManager audioMgr;
```

```
    audioMgr.play2d("soundfile.wav", /*loop*/ false);
    std::cin.get(); // prevent exiting
}
```

Code 3.4: Audio Manager example

3.2.8 OpenGL Frame Buffer Objects and the `FrameBuffer` Class

OpenGL provides Frame Buffer Objects (FBOs), which can be thought of as off-screen rendering targets—similar to virtual canvases or pseudo-windows that you can draw on. Instead of rendering directly to the screen, FBOs allow rendering to a texture or render buffer. This is particularly useful for post-processing effects, shadow mapping, and deferred rendering techniques.

The `FrameBuffer` class in the engine serves as a wrapper around OpenGL's FBO functionality, simplifying the creation, configuration, usage, and memory management of frame buffer objects. It provides functionality to bind the FBO, with the option to clear it upon binding.

For simplicity, this implementation constructs the FBO using textures (rather than **Renderbuffer Objects**), as most parts of the engine require the ability to read from the framebuffer. A depth buffer is also included. These design choices can be modified in the future to support more flexible configurations.

A simple example is provided below:

```
#include <framebuffer.h>

int main()
{
    // Ensure OpenGL context is active before using the FrameBuffer
    FrameBuffer frameBuffer;
    frameBuffer.bind();

    // All OpenGL draw calls will now render to the frame buffer
    // ...

    frameBuffer.unBind();

    // Access the color and depth texture IDs if needed
```

```
15      GLuint colorTex = frameBuffer.textureId;
16      GLuint depthTex = frameBuffer.depthTextureId;
17
18      // Or, draw the FBO's content directly to the screen
19      frameBuffer.draw();
20  }
```

Code 3.5: Frame Buffer example

3.2.9 Text And Sprite Renderers

The engine provides classes for text and sprite renderers which are both used for drawing 2d objects on the screen. Text renderer technically just draws a rectangle with blending enabled and draws text on the rectangle as a texture. Sprite renderer can be used for drawing 2d pictures on the screen. There are 2 ways of doing both of these drawings in 3D, one is to either place the 2D rectangles of these renderers always in front of the camera so they look like they are on the screen. The other simpler way is to use **Orthographic** projection instead of **Perspective** projection as the projection matrix and then draw as you normally would on a 2D screen. This second approach is what's followed in the code, however, you can follow the first approach by just changing the projection matrix. The following code snippet shows the usage of the two classes.

The sprite renderer class also comes with a `Sprite` class and all the 2D sprites must be loaded using this class – this avoids reloading the sprites unnecessarily.

```
1  #include <sprite_renderer.h>
2  #include <textrender.h>
3  #include <shader.h>
4  /*other opengl, glm includes*/
5
6  int main()
7  {
8      TextRenderer textRenderer;
9      SpriteRenderer spriteRenderer;
10
11     glm::mat4 proj = /*define orthographic proj*/
12
13     Shader textShader {/*sources to text vertex and frag shaders*/
           };
```

```
14      Shader spriteShader {/*sources to sprite vertex and frag
            shaders*/};
15
16      textShader.setMatrix("proj", proj);
17      spriteShader.setMatrix("proj", proj);
18
19      textRenderer.renderText(textShader, "some text",
20    /*x*/ 0.0, /*y*/ 0.0, /*scale*/ 1.0f,
21      /*color*/ glm::vec3(1.0f, 0.0f, 0.0f) ,
22      /*Mode*/ TextRenderer::TOP_LEFT);
23
24      //this comes from sprite_renderer
25      Sprite mySprite {/* source file */ "someFile.png",
26                       /*size*/ = 1.0f, /*rotation*/ = 0.0f,
27                       /*pulsating*/ = false};
28
29      spriteRenderer.draw(
30          mySprite, spriteShader, glm::vec3(0.0f)
31      );
32
33  }
```

Code 3.6: Text and sprite renderer example

The mode in `TextRenderer::renderText` specifies which corner of the rectangle that the text is going to be drawn on are the x, y coordinates referring to; there are five options for this: one for each of the corners, and one for the center. `SpriteRenderer::draw` also accepts rotation and mode as optional arguments. You can also optionally initialize the text renderer with a different font, the default one is Arial.

3.2.10 Architecture

The UML diagram below represents a simplified architecture of the whole game and shows how the components connect to each other.

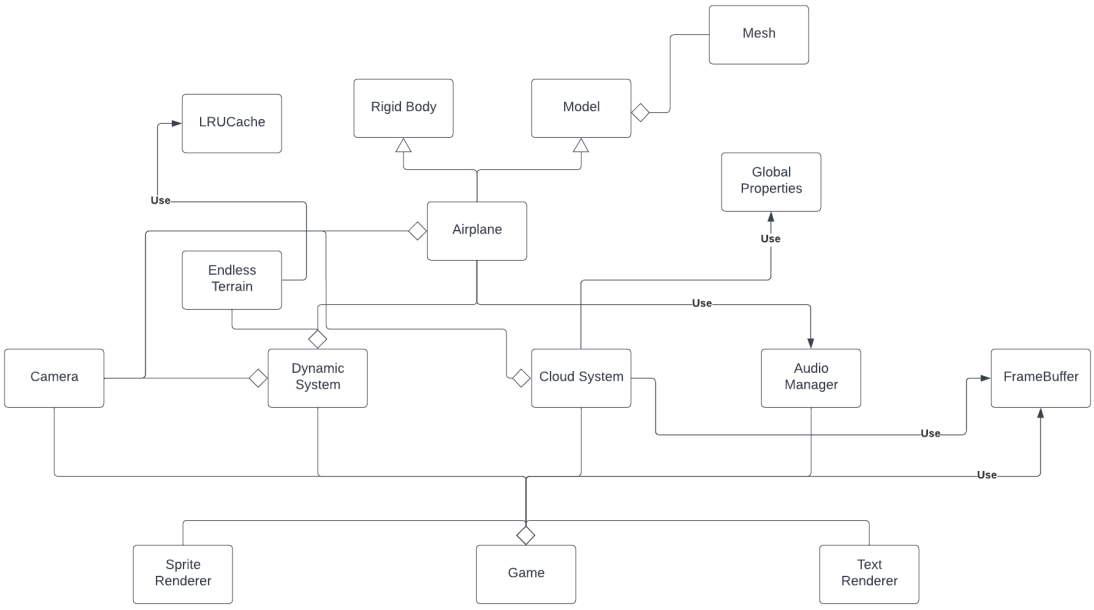


Figure 3.6: Architecture

The main components i.e endless terrain, cloud system, rigid body, and airplane are explained the following sections.

3.3 Procedural Terrain

3.3.1 Generating a chunk

Definition 2 (Chunk)**.** A *chunk* refers to a rectangular mesh of predefined size, where each vertex contains height and normal vector information. Chunks serve as the basic building blocks for terrain generation and rendering.

In most implementations where non-repetitive chunks are required, noise algorithms are commonly used. The algorithm I have chosen is **Perlin Noise** [6], which is implemented in `perlin.cpp`. Assuming, for now that we have some sort of a Perlin noise implementation available, a chunk that only has height data may be generated as simply as the following pseudo code shows:

```
def generateChunkData(size, scale):
  chunkData = ChunkData(size, size)
  for i in range(size):
    for j in range(size):
      x = j * scale
      y = i * scale
```

```
7         chunkData.height[i, j] = perlin(x, y)
8     return chunkData
```

Code 3.7: Chunk generation

Next, we will explore how to extend this basic chunk with additional data such as normals and how to tile these chunks together for a continuous terrain. We will also dive into techniques such as Level of Detail (LOD) and early culling, which help improve performance and rendering efficiency in large terrains. First, let's simplify Perlin Noise and look at how to calculate normals for our chunk data.

3.3.2 Perlin Noise

This section does not aim to provide a detailed theoretical explanation of the Perlin Noise algorithm. For a formal treatment, the reader is encouraged to refer to the original paper by Ken Perlin [6]. For a more intuitive and visual explanation, see the video referenced in [7].

Perlin Noise is a type of gradient noise that aims at generating smooth noise. If we, for instance, take a look at white noise where each pixel can be thought of as having a uniform probability of having a value between 0 and 1. The generated map as illustrated in Figure 3.7 looks completely random and this would never work as a height map as the height of a terrain is continuous.

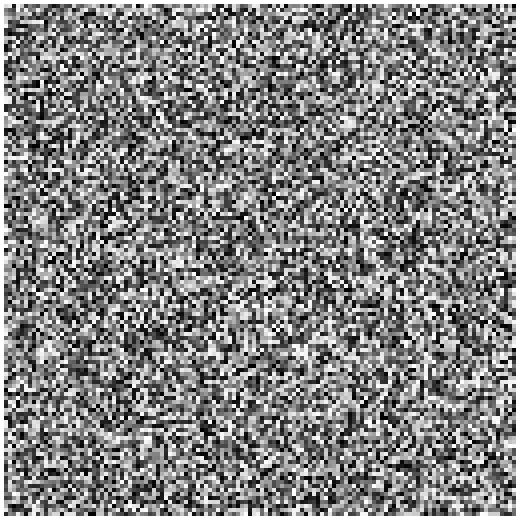


Figure 3.7: White Noise

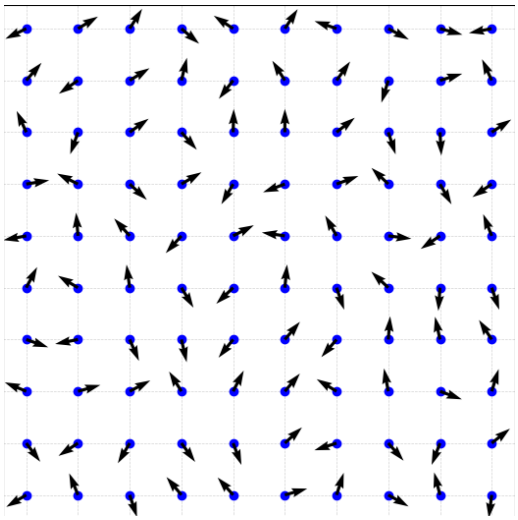


Figure 3.8: Perlin Grid

Before looking into how Perlin noise works, let's define the `lerp` function.

Definition 3 (`lerp`)**.** Lerp is defined as :

$$\texttt{lerp}(a, b, p) = a + p(b - a)$$

Perlin noise works by dividing the plane into a grid, assigning a random gradient vector at each vertex, as shown in Figure 3.8. Consider a single grid cell, and label the corners as follows:

- Top-left: P_{tl} with gradient $\vec{v}_{tl}$
- Top-right: P_{tr} with gradient $\vec{v}_{tr}$
- Bottom-left: P_{bl} with gradient $\vec{v}_{bl}$
- Bottom-right: P_{br} with gradient $\vec{v}_{br}$

Let $U = (U_x, U_y)$ be a point inside the cell, and let (u, v) be the fractional parts of (U_x, U_y) relative to the cell.

Define direction vectors from U to each corner:

$$\vec{d_{tl}} = P_{tl} - U, \quad \vec{d_{tr}} = P_{tr} - U, \quad \vec{d_{bl}} = P_{bl} - U, \quad \vec{d_{br}} = P_{br} - U$$

Next, compute the dot products:

$$G_{tl} = \langle \vec{v}_{tl}, \vec{d_{tl}} \rangle, \quad G_{tr} = \langle \vec{v}_{tr}, \vec{d_{tr}} \rangle, \quad G_{bl} = \langle \vec{v}_{bl}, \vec{d_{bl}} \rangle, \quad G_{br} = \langle \vec{v}_{br}, \vec{d_{br}} \rangle$$

A smoothing function is used to interpolate values:

$$g(t) = 6t^5 - 15t^4 + 10t^3$$

Finally, the noise value at point U is computed using bilinear interpolation:

$$\texttt{noise}(U_x, U_y) = \texttt{lerp}(\texttt{lerp}(G_{tl}, G_{tr}, g(u)), \texttt{lerp}(G_{bl}, G_{br}, g(u)), g(v))$$

The `Perlin2d` class in the engine provides an efficient implementation of the functionality described above. It also does Fractional Brownian Motion which is described in Section 3.4.5.

3.3.3 Calculating normals

In general, given a parametric surface $f(u, v)$ it is possible to obtain surface normal at specific points by the following formula:

$$\hat{n} = \frac{\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}}{\|\frac{\partial f}{\partial u} \times \frac{\partial f}{\partial v}\|}$$

One way to imagine this is to see that $\frac{\partial f}{\partial u}$ and $\frac{\partial f}{\partial v}$ span the plane that is tangent to the surface, so their cross product gives the normal vector. For a more mathematically sound argument, you can refer to Paul's Online Notes [8]. In our case the mesh is made up of triangles as seen in Figure 3.9 but we can still calculate surface normals using cross products.

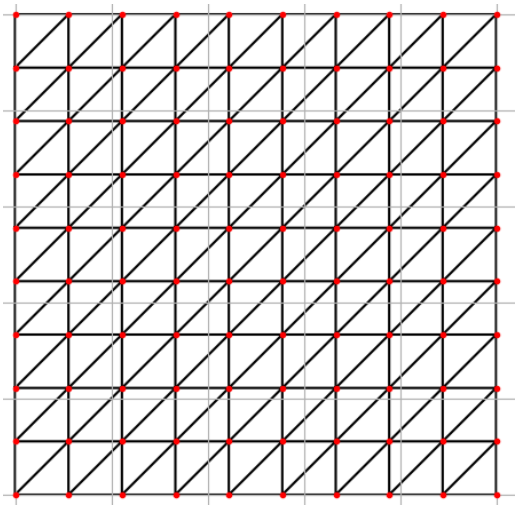


Figure 3.9: Mesh

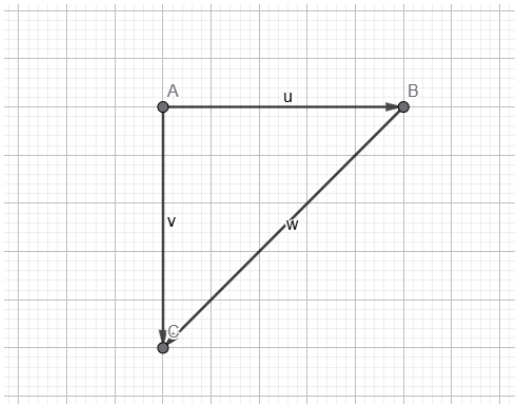


Figure 3.10: Calculating normal of a triangle

As Figure 3.10 shows we can calculate the normal at vertex **A** by calculating $u \times v$ (or $v \times u$ this direction is usually checked empirically) where u and v are the vectors of the triangle. This figure is 2d but it can be imagined the all 3 points **A**, **B**, and **C** are at different heights but the math works the same.

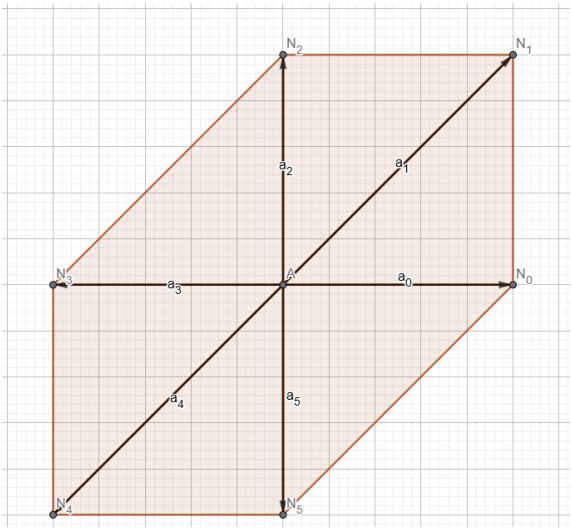


Figure 3.11: Normal Calculation

However, vertex **A** is in multiple triangles. Figure 3.11 shows my approach for calculating the normals. I take all the 6 faces (triangles) that vertex **A** is a part of. Let's call the neighboring vertices of A $\mathbf{N_0}...\mathbf{N_5}$ in either counterclockwise or clockwise ordering (depending on which way defines the cross products in the up direction – this needs to be checked empirically). Then the normal vector can be calculated as follows:

$$\vec{n} = \frac{1}{6}\sum_{i=0}^{i=5}(\mathbf{N_i} - \mathbf{A}) \times (\mathbf{N}_{(\mathbf{i+1}) \bmod \mathbf{6}} - \mathbf{A})$$

Which is the average of the the normal vectors for each of the 6 triangles. For this to work correctly, you must ensure the vertices are in the correct ordering.

As the picture above suggests, this breaks down at the edges of the mesh. To circumvent this issue, I pad the mesh with extra set of vertices that is used just for the calculation of normals.

So, the pseudocode with normals looks something like this:

```
def generateChunkData(size, scale):
  heightData = Matrix(size + 2, size + 2) // matrix with num rows,
     cols = size + 2
  for i in range(size + 2):
    for j in range(size + 2):
      x = j * scale
      y = i * scale
      heightData[i, j] = noise(x, y)

```

```
9   chunkData = ChunkData(size, size)
10  for i in range(1, size):
11    for j in range(1, size):
12      vs = getNeighbors(i, j) //should return list of 4  neighbors
            in the correct order
13      vertex = Vector(i, heightData[i, j] ,j)
14      normal = Vector(0, 0, 0)
15      for k in range(6):
16        a = vs[k]
17        b = vs[(k + 1) % 6]
18        normal += cross(a-vertex, b-vertex)
19      mag = normal.magnitude()
20      if mag != 0:
21        normal /= mag
22      chunkData[i-1, j-1].height = heightData[i, j]
23      chunkData[i-1, j-1].normal = normal
24  return chunkData
```

Code 3.8: Chunk generation Part 2

3.3.4 Geometry Shaders and verifying the correctness of the normals

It still needs to be checked if the normals calculated are actually correct and one way of doing that is to visualize them using geometry shaders. A tutorial for which can be found on LearnOpenGL [5]. The `Shader` class provided with the engine takes an optional third argument -the geometry shader source file- so, integrating it into the existing system is easy.

A geometry shader can take primitives as input (a triangle in our case) and can output primitives. For each vertex of the triangle we can emit a line with two vertices one at the original vertex, the other one in the direction of the normal but scaled by some constant α.

The scene must be rendered twice once with the normal shader and again with the geometry shader. Figure 3.12 was obtained using this pipeline:

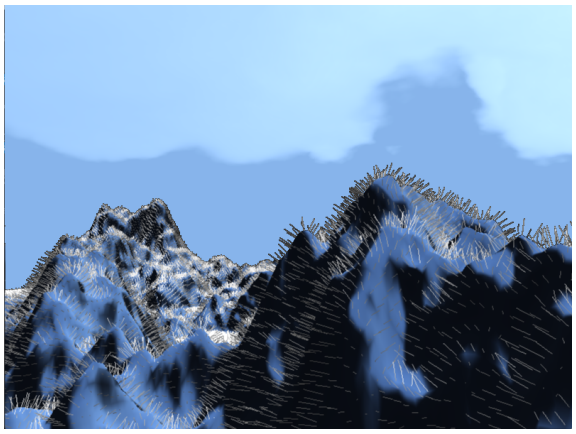


Figure 3.12: Visualizing Terrain Normals

3.3.5 Tiling the chunks

Once we have the chunk generation function above it's not too hard to tile them. Since Perlin Noise accepts (x, y) as the arguments we just need to make sure when going from one chunk to the next we give the correct coordinates to Perlin Noise. To do this, I extended the chunk generation function to accept one more argument: its `center`, using this it calculates the noise as it previously did from its top left corner but this time shifted by the `center` coordinates.

The following pseudocode explains how it is done:

```
def generateChunkData(size, center, scale):
  heightData = Matrix(size + 2, size + 2) // matrix with num rows,
     cols = size + 2
  tlX = (size - 1)/-2.0 //top left x
  tlY = (size - 1)/2.0 //top left y
  for i in range(-1, size + 1):
    for j in range(-1, size + 1):
      x = (center.x + tlX + j) * scale
      y = (center.y + tlY - i) * scale
      heightData[i, j] = noise(x, y)

  //rest remains same
```

Code 3.9: Chunk generation Part 3

If the loop started from `0`, then `center.x + tlX` would give the x-coordinate of the current chunk. However, at index $(0, 0)$, we actually want to sample from the previous chunk's coordinate space due to the padding added for normal calculation.

Therefore, the loop must begin at `-1` to correctly align the noise sampling with the padded grid.

3.3.6 Level Of Detail (LOD)

It's important for a better FPS to render the chunks near the player at a higher level of detail than those that are further away. LOD can be thought of in this case as the number of vertices in a given area or volume.

To do this I first create a mesh and then further lower resolution meshes are created by skipping some vertices in the original mesh such that the first and the last vertices are not skipped because the mesh must extend from and to the same points. This begs the question then what should be the size of the original mesh? Consider a 1d mesh of size 10 as the picture below 3.13 illustrates. We cannot skip every 2nd vertex because then we will have $\langle 0, 2, 4, 6, 8 \rangle$ and the last 9th vertex would be skipped.

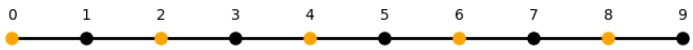


Figure 3.13: 1D mesh

So, if k is the skip factor and s is the size of the mesh, then for us to be able to create a mesh, k must divide $s - 1$. We need to find s such that $s - 1$ has a lot of divisors, and a commonly chosen number is 241 since every number from 1 to 6 divides 240 so we can create a mesh that has $\frac{1}{6}$ of the vertices in the original mesh.

A common approach for this is to generate these meshes on the fly as a function of the distance from the player's coordinates. However, I found this method to not match my FPS goals. So, what I do instead is create the 6 LOD meshes beforehand and give the corresponding mesh its height and normal data through a texture. Height and normal data are stored in a texture, where both of them are packed into a `vec4` using the red, green, blue, and alpha channels. Each mesh then simply samples from this texture to obtain terrain shape and lighting information.

Unlike traditional LOD systems that generate meshes at runtime based on camera distance, this approach precomputes multiple LOD meshes and uses texture data

to drive the final geometry. This reduces runtime computation and improves FPS stability at the cost of some memory overhead.

The image 3.14 below illustrates how **LOD** meshes look like in practice.

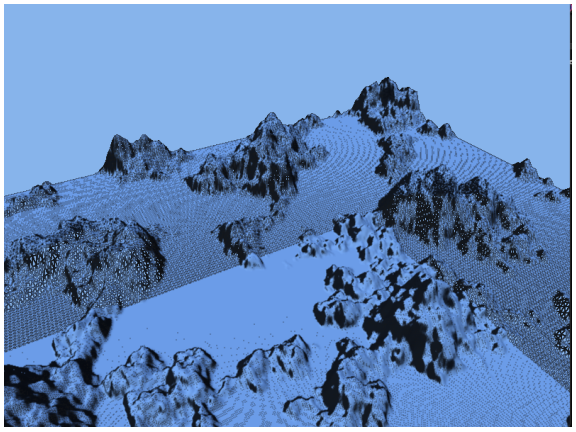


Figure 3.14: Visualizing LOD Terrain

3.3.7 Multithreading

It is not feasible to draw and generate new chunks on the same thread; for instance, we may need to generate 9 chunks at once, which would block the main render loop and drop the frame rate. To avoid this, a new thread is dispatched for generating data for each chunk. Since the meshes are already pre-generated (as discussed in the LOD section), the data needed is just height and normals. Figure 3.15 illustrates this pipeline.

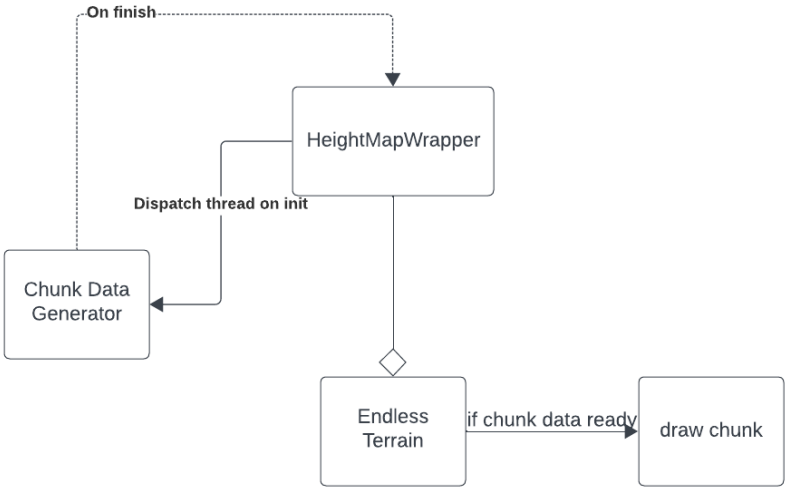


Figure 3.15: Chunk Drawing Pipeline

The `EndlessTerrain` class only draws a chunk once its data is ready. When a chunk is needed, it initializes a `HeightMapWrapper` instance, passing in the chunk center. This wrapper then launches a new thread to request height and normal data from the `ChunkDataGenerator` (which is just a simple function in the code). Once the data is computed, it is returned via a callback function, completing the pipeline. I used Barak Shoshany's thread pool library [9] for managing the threads.

IMPORTANT:

It is crucial to note that OpenGL operates strictly on a single thread—all OpenGL commands must be issued from the same thread that initialized the OpenGL context. Because of this, the `HeightMapWrapper` class handles all computation on the CPU first. Once the data is available (from the background thread), it schedules OpenGL commands to upload the data to the GPU as a texture, all on the main thread.

3.3.8 Caching and the `LRUCache` class

Imagine we generate 9 chunks around the player for a single frame, to avoid re-generating them in the next one we must cache them. A straightforward solution would be to use a hashmap where the key is the center of the chunk and the value is the corresponding height map. However, this approach leads to an unbounded cache, which can eventually exhaust system memory.

To solve this, I used the `LRUCache` class that comes with the engine—a bounded version of the classic hashmap. If the maximum cache size is n and a new key is inserted when the cache is full, the least recently used (LRU) key is evicted to make space.

Since LRUCache is a well-known data structure, its full implementation details are omitted. However, figure 3.16 illustrates its core design using a doubly linked list, which should provide enough insight into how the code operates.

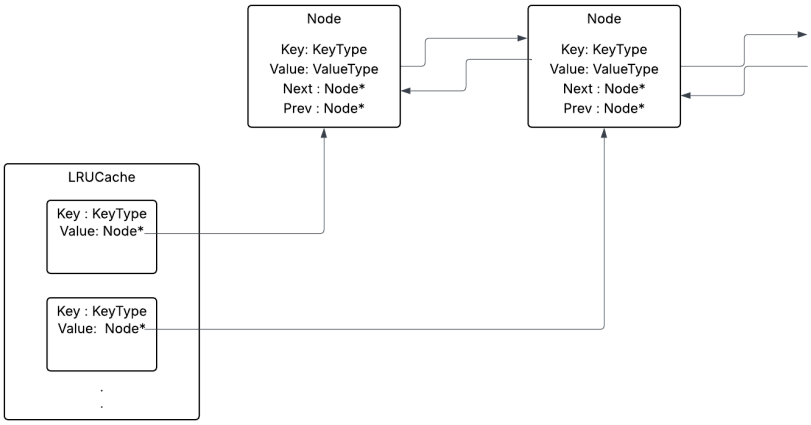


Figure 3.16: LRUCache design

3.3.9 Early Culling

Notice that the methods above issue generate and draw commands for chunks that are not in the view frustum of the player. This introduces an overhead that can be easily avoided. For instance, the chunks that are behind the player can be omitted. To address this, I use a simple directional culling approach: Say, $\vec{d}$ is the direction of the camera and $\mathbf{P}$ is its position, when we are generating the i_{th} chunk with center $\mathbf{C_i}$ around the player, let $\vec{v} = \mathbf{P} - \mathbf{C_i}$ be the direction of the chunk. We only generate data for the chunk if $\langle \vec{d}, \vec{v} \rangle > 0$ that is, both of those vectors point in the same direction. Note that if $\vec{v}$ is 2 dimensional that $\vec{d}$ must be projected on to the plane. In the code provided, $\mathbf{P}$ is projected onto the $x - z$ plane and so is $\vec{d}$.

Result: This optimization led to a significant performance gain—FPS improved, and memory usage dropped from around 500 MB to around 100MB.

3.4 Volumetric Clouds

This section explains my implementation of volumetric clouds. Before delving into the details, I would like to acknowledge the resources that were instrumental in my work and thank the online community: [1] [2] [10] [3] [11] [12] [13] [14] [15].

3.4.1 Ray Marching

Before rendering clouds, it's essential to understand the basics of ray marching. Ray marching is a technique where we trace rays for each pixel on the screen. Unlike

ray tracing, which requires information about polygonal geometry, ray marching only requires access to a signed distance function (sdf).

A signed distance function for an object O and a point P returns the distance from P to the nearest point on O's surface. It is:

- positive if P is outside the object
- negative if P is inside
- and zero if P lies on the surface

For example, the signed distance function for a sphere is:

$$\text{sdf}(p) = \|p - c\| - r$$

where c is the center and r is the radius of the sphere.

A ray is defined by its origin P and direction $\vec{d}$. We represent it as a function of time:

$$R(t) = P + \vec{d}t$$

If there are multiple objects in the scene, the scene's overall signed distance function is defined as:

$$\text{sdf}_{\text{world}}(P) = \min_{\text{objects}} \text{sdf}_{\text{object}}(P)$$

Ray marching works by advancing the ray by the distance given by the sdf at each step, as illustrated in the following code snippet:

```
int MAX_STEPS = 1000;
float MIN_DIST = 1e-4;
float MAX_DIST_TRAVEL = 1e3;

vec3 trace(Ray r) // returns light info
{
    vec3 currentPos = r.pos;
    float totalDistTraveled = 0.0;

    for (int i = 0; i < MAX_STEPS; ++i)
    {
        currentPos = r.pos + r.direction * totalDistTraveled;
        float dist = sdf_world(currentPos);

        if (dist < MIN_DIST) {
```

```
            // do light calculation
            return /* light info */;
        }

        if (totalDistTraveled > MAX_DIST_TRAVEL) break;

        totalDistTraveled += dist;
    }
    return vec3(0.0);
}
```

Code 3.10: Basic Ray Marching

The illustration in Figure 3.17 shows this process in 2D, which can be interpreted as a top-down view of a 3D scene.

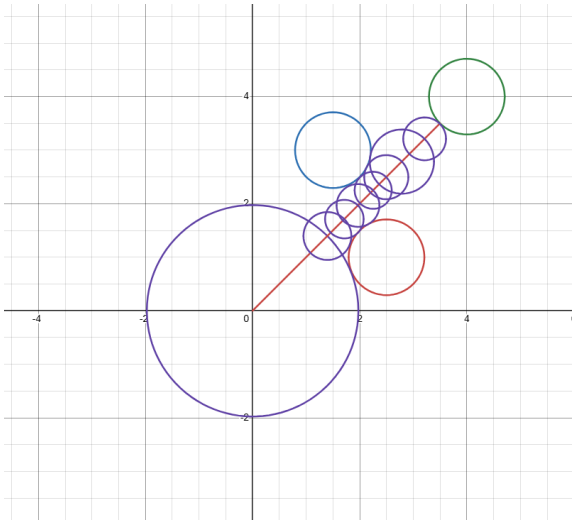


Figure 3.17: Ray marching example

When $\text{sdf}(p) = 0$, the surface near point p can be locally approximated by the equation $\text{sdf}(p) = 0$. For an implicit surface $f(x, y, z) = c$ (which can be thought of as a equipotential surface in 3D or level curve in 2D), the surface normal is given by the gradient:

$$\nabla f = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right\rangle$$

Using a symmetric finite difference approximation:

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}$$

we can approximate the normal at point p from the signed distance function as:

$$\vec{n} = \left\langle \text{sdf}(p + h\hat{i}) - \text{sdf}(p - h\hat{i}), \text{sdf}(p + h\hat{j}) - \text{sdf}(p - h\hat{j}), \text{sdf}(p + h\hat{k}) - \text{sdf}(p - h\hat{k}) \right\rangle$$

This normal vector is then used in lighting calculations.[1]

3.4.2 Ray Marching for rendering volumes

For rendering volumetric surfaces like clouds instead of using *sdf* functions a fixed step size δ is chosen and the ray is moved forward by this step size at each step. At each point P in space density is sampled and the rendering is done based on how much density was observed, how much light passed through and so on. More details are in the following section but first for clouds my implementation makes a rectangular area in the sky where the clouds can be, this is a axis aligned bounding box (AABB). First, a check is done to see if the ray intersects this box; if yes, only then is the ray marched through the box. Next, we derive the intersection of a ray with an AABB. Remember that a ray is given as $R(t) = P + \vec{dt}$. An axis aligned bounding box can be defined by its minimum and maximum coordinates denoted as b_{min}, b_{max}. The ray will intersect the box if:

$$b_{min} \leq P + \vec{dt} \leq b_{max}$$

$$b_{min} - P \leq \vec{dt} \leq b_{max} - P$$

which gives 3 inequalities:

$$\frac{(b_{min,x} - P_x)}{\vec{d}_x} \leq t \leq \frac{(b_{max,x} - P_x)}{\vec{d}_x}$$

$$\frac{(b_{min,y} - P_y)}{\vec{d}_y} \leq t \leq \frac{(b_{max,y} - P_y)}{\vec{d}_y}$$

$$\frac{(b_{min,z} - P_z)}{\vec{d}_z} \leq t \leq \frac{(b_{max,z} - P_z)}{\vec{d}_z}$$

if $[t_{min}, t_{max}]$ is the intersection of those intervals, then the ray intersects the box if $t_{max} \geq t_{min}$ with t_{min} being the nearest point of intersection and t_{max} being the furthest. If the point is inside the box then clearly, $t_{min} < 0$ in this case we return

[1] This reasoning was derived independently (so it's more intuitive than rigorous) but is consistent with standard SDF-based normal estimation techniques used in ray marching.

t_{max} as the intersection t. if $t_{max} < 0$, the box must be behind the ray, so there is no intersection.

3.4.3 Worley Noise

Worley noise is a type of cellular noise that we are going to use for generating density textures for clouds. Typically, Worley noise works by choosing n random feature points, in my code I have taken the liberty to call them anchor points. Then for a pixel at location p we calculate its value:

$$v_p = \min_{anchors} \|p - a_i\|$$

v_p for all pixels should be normalized such that $0 \leq v_p \leq 1$. Figure 3.18 visualizes how this looks like.

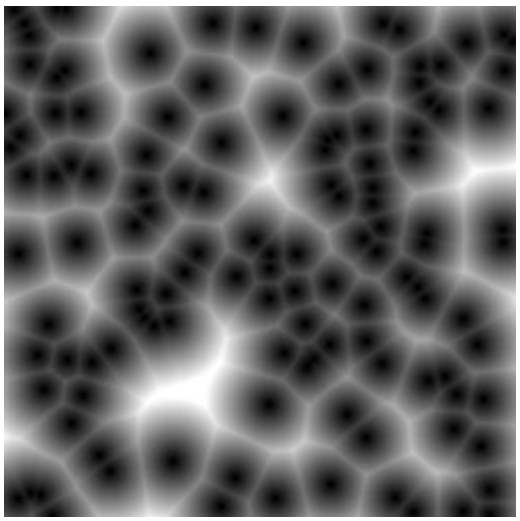


Figure 3.18: Worley Noise [16]

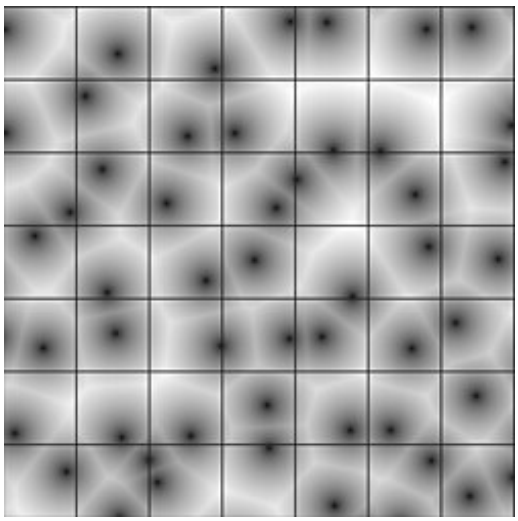


Figure 3.19: Grid Based Worley Noise [16]

However, the naïve approach is computationally expensive, since it requires computing distances from each pixel to all anchor points—scaling poorly as the number of anchors increases. To make it efficient, a grid-based approach is used: the space is divided into regular grid cells, and one anchor point is placed in each cell.

Now, the nearest anchor point for any given pixel can only lie within its own grid cell or one of the adjacent cells. This reduces the number of distance checks significantly.

Normalization becomes straightforward as well: in 2D, the maximum possible distance to a corner in a unit grid cell is $\sqrt{2}$ (assuming the grid cells are of unit

area), so the normalized score s_p is:

$$s_p = \frac{v_p}{\sqrt{2}}$$

Similarly, in 3D, the normalization factor becomes $\sqrt{3}$:

$$s_p = \frac{v_p}{\sqrt{3}}$$

A more useful version in our case is **Inverted Worley Noise** which is calculated as $1 - s_p$. This gives the bubbly shapes that clouds have as the figure 3.20 below illustrates:

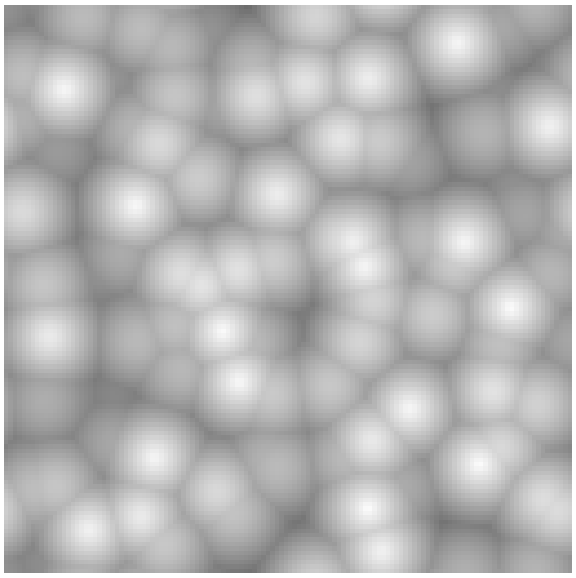


Figure 3.20: Inverted Worley Noise

In the following sections when I say Worley noise, I mean Inverted Worley Noise unless otherwise stated.

3.4.4 Making Tileable Worley Noise

As we will see in the following sections, Inverted Worley noise is used for generating 3D density textures which must be tileable. This means for a pixel on the edge we must take into consideration the cell that is on the other edge (imagine wrapping the grid around like a torus) and the anchor point it contains. My approach for this was to use what I called a deterministic random function which always produces the same value for the same input. The code snippet below shows how it works:

```
#returns anchor points for grid cell at position pos
def det_random(pos : tuple):
  random.seed(hash(pos))
  return [random.random() for i in range(len(pos))]
```

Code 3.11: Deterministic Random Function

Given some grid coordinates this function generates the anchor point for that cell, even though the anchor point itself is random, it will always give the same anchor point for a given grid cell (in that sense it is deterministic). This way we don't have to store these anchor points. Then we can get the score (or the color) for a pixel as follows:

```
//returns anchor points for grid cell at position pos
NUM_TILES  = #number of tiles (boxes)
BOX_WIDTH = WIDTH / NUM_TILES
BOX_HEIGHT = HEIGHT / NUM_TILES
MAX_DIST = sqrt(BOX_WIDTH * BOX_WIDTH + BOX_HEIGHT * BOX_HEIGHT)
def get_color(x, y):
  color = 0.0
  min_d = float('inf')

  #get tile coordinates
  tileX = x // BOX_WIDTH
  tileY = y // BOX_HEIGHT

  #iterate over neighbor tiles
  for i in range(-1, 2):
    for j in range(-1, 2):
      neighbor_y = (tileY + i) % NUM_TILES
      neighbor_x = (tileX + j) % NUM_TILES
      anchor_offsetX, anchor_offsetY = det_random((neighbor_x,
          neighbor_y))

      #rescale to actual coordinates, note these are not wrapped
      anchor_x = (tileX + j + anchor_offsetX) * BOX_WIDTH
      anchor_y = (tileY + i + anchor_offsetY) * BOX_HEIGHT

      min_d = min(min_d, dist((x, y), (anchor_x, anchor_y)))
  color = min_d/MAX_DIST
  #inverted Worley
```

```
28   return (1 - color)
```

Code 3.12: Tileable Worley Noise

All textures used in the game were pre generated using Python and the corresponding code can found can be found in the `noise_scripts` subdirectory.

3.4.5 Fractional Brownian Motion

Nearly all noise functions used in procedural generation benefit from a technique called **Fractional Brownian Motion** (fBM). It enhances basic noise by layering multiple octaves—copies of the noise function with increasing frequency and decreasing amplitude—to produce a more complex, self-similar structure reminiscent of fractals. This gives a more natural, detailed appearance when zoomed in.

The following code shows a typical fBM implementation:

```
1  FREQ_MULT = 2.0
2  AMPL_MULT = 0.5
3
4  def fbm(x, y, octaves):
5    ampl = 1 #amplitude
6      freq = 1 #fequency
7      value = 0.0
8      max_val = 0.0
9
10     for i in range(octaves):
11         value += ampl * noise(x, y)
12         max_val += ampl
13         ampl *= AMPL_MULT
14         x *= FREQ_MULT
15         y *= FREQ_MULT
16
17     return value/MAX_NOISE_VAL # normalized
```

Code 3.13: Fractional Brownian Motion

Imagine a plot of a signal, the code above adds higher and higher frequency values of the signal with lower amplitudes to itself to make it appear self-similar.

3.4.6 Perlin Worley Noise

The inverted Worley noise is combined with Perlin noise to add detail to the low-density regions of Perlin noise, creating textures that result in puffier-looking clouds—a technique originating from Guerrilla Games [10]. Before presenting the formula used to combine the two types of noise, we first define the remap function.

Definition 4 (Remap function)**.**

$$\texttt{remap}(a, old\_min, old\_max, new\_min, new\_max) =$$

$$new\_min + (new\_max - new\_min)\frac{a - old\_min}{old\_max - old\_min}$$

Let p represent Perlin noise and w the inverted Worley noise. The Perlin-Worley noise is computed as:

Definition 5 (Perlin Worley Noise)**.**

$$\texttt{PerlinWorley}(p, w) \ = \texttt{remap}(p, 0.0, 1.0, w, 1.0)$$

The figures below shows how FBM worley noise combined with Perlin noise looks like.

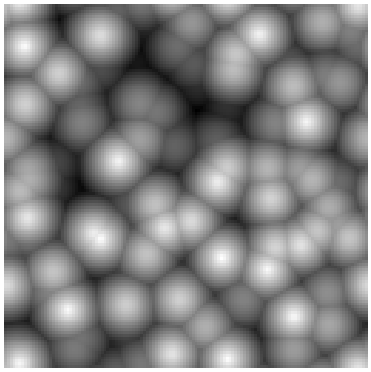


Figure 3.21: Worley Noise

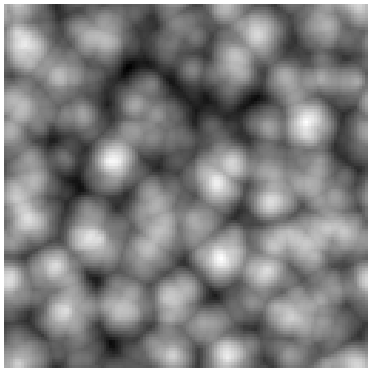


Figure 3.22: FBM Worley Noise

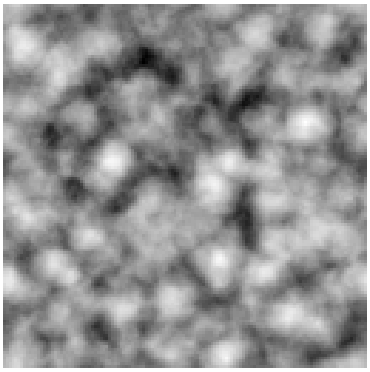


Figure 3.23: Perlin Worley Noise

3.4.7 Textures And Sampling Density For Clouds

One of the 3D textures for clouds contains Perlin-Worley noise in the red channel, and Worley noise of increasing Frequencies in the green, blue, and alpha channels, this is referred to as the low frequency noise texture and its size is $128 \times 128 \times 128$. The base cloud is obtained as follows [15]:

```
float getCloudDensity(vec3 p)
{
    /*
    normalize p before hand
    to be able to sample nicely from the texture,
    in the code, this is usually done by multiply
    p with a uniform scale variable.
    */
    vec4 low_freq = texture(cloudTexture, p);
    float wfbm = 0.625 * low_freq.g + 0.25 * low_freq.b + 0.125 *
        low_freq.a;
    float baseCloud = Remap(low_frequency_noises.r, wfbm-1.0, 1.0,
        0.0, 1.0);

    return baseCloud;

}
```

Code 3.14: Base Cloud

Another 3D texture referred to as high frequency noise texture contains Worley noise of increasing Frequencies, its size is $32 \times 32 \times 32$, and this is used for adding details on the edges of the clouds – this step can be optimized, as we only need to add details to those clouds that are near a certain threshold to the camera.

The last texture that I use is called a weather texture that contains following values [15]:

- Cloud Coverage: The percentage of cloud coverage in the sky
- Precipitation: The chance that clouds will rain (This is not used in my code)
- Cloud Type: A value of 0.0 indicates stratus, 0.5 indicates stratocumulus, and 1.0 indicates cumulus clouds.

I found the following code snippet on [12] for getting the density of of a cloud based on its type which is given by the weather map:

```

#define STRATUS_GRADIENT vec4(0.0, 0.1, 0.2, 0.3)
#define STRATOCUMULUS_GRADIENT vec4(0.02, 0.2, 0.48, 0.625)
#define CUMULUS_GRADIENT vec4(0.00, 0.1625, 0.88, 0.98)
```

```
5
6  float getDensityForCloudOfType(float heightFraction, float
       cloudType){
7      float stratusFactor = 1.0 - clamp(cloudType * 2.0, 0.0, 1.0);
8      float stratoCumulusFactor = 1.0 - abs(cloudType - 0.5) * 2.0;
9      float cumulusFactor = clamp(cloudType - 0.5, 0.0, 1.0) * 2.0;
10
11     vec4 baseGradient = stratusFactor * STRATUS_GRADIENT +
           stratoCumulusFactor * STRATOCUMULUS_GRADIENT + cumulusFactor
            * CUMULUS_GRADIENT;
12
13     float result = remap(heightFraction, baseGradient.x,
           baseGradient.y, 0.0, 1.0) * remap(heightFraction,
           baseGradient.z, baseGradient.w, 1.0, 0.0);
14     return result;
15 }
```

Code 3.15: Calculating Specific Cloud Density

To understand what this function is doing, we imagine that the cloudType is 0, in which case the baseGradient will just be STRATUS_GRADIENT which defines how the density for the stratus cloud should be. Since stratus clouds are at lower height, the first remap in line 13 is saying: the density should increase from 0.0 to 0.1 height, the second remap is saying: the density should decrease from height 0.2 to 0.3, their multiplication gives a gradient (a quadratic function) that is plotted in the image 3.24 below:

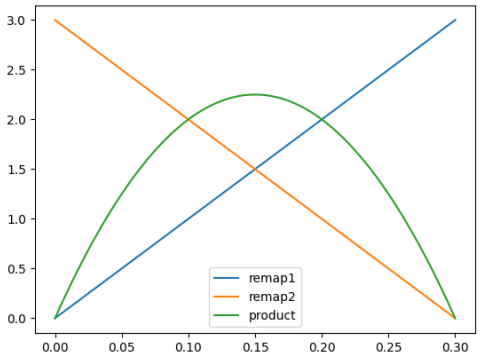


Figure 3.24: Height Gradient of Stratus Cloud

The base cloud above is then enhanced the following way [15] [12]:

```
float getCloudDensity(vec3 p)
{
    /*baseCloud same as above
    boundsMin, boundsMax are the min, max coords of the bounding
       rectangle (AABB)
    */

    /*this line is not needed if your coverage texture is good
       enough.*/
    baseCloud = max(baseCloud-densityThreshold, 0.0)

    heightFraction = (p.y - boundsMin.y)/(boundsMax.y-boundsMin.y)
    weatherData = getWeatherTexture(p); //get data from weather
       texture

    float heightGradient = getDensityForCloudOfType(weatherData.r);
    baseCloud *= heightGradient;

    float coverage = clamp(weatherData.g, 0., 1.) *
       coverageMultiplier;
    float coverageCloud = remap(baseCloud, coverage, 1., 0., 1.);
    coverageCloud *= coverage;
    float finalCloud = coverageCloud;

    if (finalCloud > 0)
    {
        /*make sure to scale p here too*/
        high_freq = texture(cloudHighTexture, p);
        float highFBM = high_freq.r * .625 + high_freq.g * .25 +
           high_freq.b * .125;
        /*introduce some height based turbulence*/
        float highModifier = mix(highFBM, 1.0-highFBM, saturate(
           heightFraction * 10.0));
        /*add details to edges of the cloud (where density is lower
           )*/
        finalCloud = remap(finalCloud, highModifier*0.2, 1.0, 0.0,
           1.0);
    }
    return finalCloud;
}
```

Code 3.16: Density For Cloud

3.4.8 Lighting

There is only one light source in the code: the sun. Its direction $\vec{w}_{sun}$ is a global parameter, note that this vector defines direction to the sun and not the other way around. To have accurate lighting for the clouds we need to correctly model how much light they transmit. The most thorough derivations of light calculation I could find were in Juraj Páleník's [13] thesis work, [1] [3] also seemed to use similar calculations which were instrumental in my understanding of these equations.

When light hits a particle the following things may happen [13]:

- It may get absorbed by the particle and no longer contribute to lighting effects.
- The light may get emitted by matter due to Planck's black body radiation.
- The light may get scattered in a different direction.

Given a point x and a direction $\vec{w}$, $L(x, \vec{w})$ defines the intensity of the light at point x in the direction $\vec{w}$ (Here $\vec{w}$ is the direction that you shoot your ray in for ray marching). The following differential equation defines the above steps mathematically [13]:

$$\frac{\partial L(x, \vec{w})}{\partial x} = -(\alpha(x) + \sigma(x))L(x, \vec{w}) + L_e(x, \vec{w}) + L_i(x, \vec{w})$$

Where $\alpha(x)$ is the light that is absorbed away and $\sigma(x)$ is the light that is scattered, this loss is proportional to how much light there was in the first place. $L_e(x, \vec{w})$ and $L_i(x, \vec{w})$ are the factors for light emitted in at point x in the direction $\vec{w}$ and the light scattered in the direction of $\vec{w}$ which is commonly referred to as in-scattering.

Now, the differential equation above is a first order equation (assuming $\vec{w}$ is constant) and we can put it in the standard form:

$$y' + p(x)y = q(x)$$

which can be solved using the integral factor: $u = e^{\int p(x)dx}$. Let's take $p(x) = (\alpha(x) + \sigma(x))$ and $q(x) = L_e(x, \vec{w}) + L_i(x, \vec{w})$.

We get:

$$(L(x,\vec{w})e^{\int p(x)dx})' = e^{\int p(x)dx}q(x)$$

$$L(x,\vec{w}) = e^{-\int p(x)dx}\int e^{\int p(x)dx}q(x)dx + Ce^{-\int p(x)dx}$$

If b is the starting point then:

$$L(a,\vec{w}) = e^{-\int_b^a p(x)dx}\int_b^a e^{\int_x^a p(t)dt}q(x)dx + Ce^{-\int_b^a p(x)dx}$$

Where $C = L_b = L(b,\vec{w})$. L_b represents the incoming light intensity at the starting point b — for instance, this may be the background color or sky light depending on the implementation.

$$L(a,\vec{w}) = e^{-\int_b^a p(x)dx}\int_b^a e^{\int_x^a p(t)dt}q(x)dx + L_b e^{-\int_b^a p(x)dx}$$

Rewriting the outer integral to carry out the multiplication:

$$L(a,\vec{w}) = e^{-\int_b^a p(t)dt}\int_b^a e^{\int_x^a p(t)dt}q(x)dx + L_b e^{-\int_b^a p(x)dx}$$

$$L(a,\vec{w}) = \int_b^a e^{-\int_b^a p(t)dt+\int_x^a p(t)dt}q(x)dx + L_b e^{-\int_b^a p(x)dx}$$

$$L(a,\vec{w}) = \int_b^a e^{-\int_b^x p(t)dt}q(x)dx + L_b e^{-\int_b^a p(x)dx}$$

Adapting the convention from [13], let $\tau(a,b) = \int_a^b p(t)dt$. The equation then becomes:

$$L(a,\vec{w}) = \int_b^a e^{-\tau(b,x)}q(x)dx + L_b e^{-\tau(b,a)}$$

To keep the equations same as in the literature, I will write it as:

$$L(a,\vec{w}) = \int_b^a e^{-\tau(x,a)}q(x)dx + L_b e^{-\tau(b,a)} \tag{3.2}$$

which is the same as above but we are integrating τ in the forward direction in the integral since we start at b and end at a.

As noted in [13], the exact form of L_i is the integral of light scattering from all directions (To be precise, this is a surface integral of probability of light scatter-

ing times the distribution in which the light scatters in given directions times the intensity of light):

$$L_i(x, \vec{w}) = \frac{1}{4\pi} \oint \sigma(x, \vec{w}) \tilde{p}(x, w, w') I(x, w, w') d\Omega'$$

Where $\tilde{p}$ is called the phase function which gives a distribution of redirection of light. Most implementations use the Henyey–Greenstein phase function to model the scattering directionality, which depends on the cosine of the angle between the incoming light direction $\vec{w}_{sun}$ and the viewing direction $\vec{w}$. The exact formula is omitted here as it does not provide significant insight, but it is implemented in the code and widely documented in the literature and for further mentions of it we are going represent it as $\tilde{p}(\vec{w}, \vec{w}_{sun})$.

Now we make some simplifications to evaluate this numerically and let's say $\rho(x)$ is the density at point x: [13]

- This was previously mentioned but is worth reiterating: There is only one light source: the sun.
- $L_e(x) = 0$ - there is no emission from the clouds.
- $\alpha(x) = 0$ - no light is absorbed.
- $\sigma(x) = \sigma\rho(x)$ - the light that is scattered away is proportional to the density of the cloud at that point.
- $L_i(x, \vec{w}) = \sigma\rho(x)\tilde{p}(\vec{w}.\vec{w}_{sun})e^{-\int_c \sigma\rho(s)ds}$ - this is a practical approximation of the in-scattered light. The integral is computed along the ray from point x to the sun and represents the attenuated intensity of sunlight at x, according to Beer's law, which states that light intensity decays exponentially as it travels through a participating medium.

Substituting the above simplifications into Equation 3.2 we get:

$$L(a, \vec{w}) = L_b e^{-\tau(b,a)} + \int_b^a e^{-\tau(x,a)} L_i(x, \vec{w}) dx$$

and

$$\tau(a, b) = \int_a^b \sigma\rho(x) dx$$

$$L(a,\vec{w}) = L_b e^{-\tau(b,a)} + \int_b^a e^{-\tau(x,a)}\sigma\rho(x)\tilde{p}(\vec{w}.\vec{w}_{sun})e^{-\int_c \sigma\rho(s)ds}dx$$

Making it discrete, and assuming the integral from b to a is divided into N steps:

$$L(a,\vec{w}) = L_b e^{-T_N} + \sigma\tilde{p}\sum_{i=0}^{N-1} e^{-T_i}\rho_i e^{-T_i'}\Delta x$$

Where

$$T_i = \sum_{j=0}^{i}\sigma\Delta x\rho_j$$

And,

$$T_i' = \sigma\Delta s\sum_{j=1}^{k}\rho(\mathbf{x_i} + \vec{w}_{sun}j\Delta s)$$

which is accumulating the density towards the sun, and in practice k is around 6 as mentioned in [10]. It is more useful to write these equations in a recursive format which is especially useful for programming them. The conventions are obtained from [13].

The illumination from sun can be written as (this is the $e^{-T_i'}$ part of the above equation):

$$\mathcal{I}(\mathbf{x_i}) = \mathcal{I}_i = e^{-\sigma\Delta s\sum_{j=1}^{k}\rho(\mathbf{x_i}+\vec{w}_{sun}j\Delta s)}$$

The other part of the sum (e^{-T_i}) can be written as (this is called the extinction factor):

$$\mathcal{E}_n = e^{-\sigma\Delta x\sum_{j=0}^{n}\rho_j} = \prod_{j=0}^{n} e^{-\sigma\Delta x\rho_j}$$

Giving the recurrence:

$$\mathcal{E}_0 = 1$$

$$\mathcal{E}_n = e^{-\sigma\Delta x\rho_j}\mathcal{E}_{n-1}$$

The second part of the sum for L can now be written as $C_n = \sigma\tilde{p}\Delta x\sum_{i=0}^{n}\mathcal{E}_i\rho_i\mathcal{I}_i$

Giving the recurrence:

$$C_0 = 0$$

$$C_n = \sigma\tilde{p}\Delta x\mathcal{E}_n\rho_n\mathcal{I}_i + C_{n-1}$$

Giving the equation:

$$L = L_b\mathcal{E}_N + C_N$$

3.4.9 The Complete Pipeline

The terrain is rendered first to a framebuffer object (FBO). The cloud rendering pipeline then uses the textures from this FBO as input, layering volumetric clouds on top of the scene — a process commonly known as deferred shading.

A compute shader is used to cast rays from every pixel on the screen. Rays that do not intersect the cloud bounding volume are terminated early for efficiency. For rays that do intersect, the shader performs ray marching through the cloud volume and accumulates light based on the equations discussed previously.

Finally, the result of the compute shader is drawn to the screen.

The figures below show the clouds enclosed within their bounding volume, from both top and bottom perspectives.

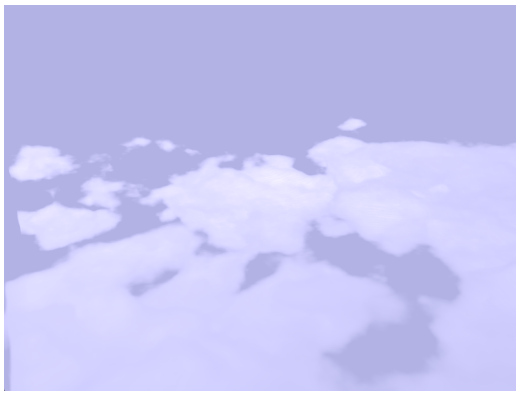


Figure 3.25: Clouds top view

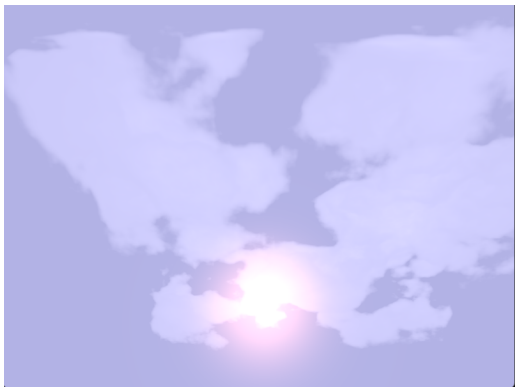


Figure 3.26: Clouds bottom view

3.5 Rigid Body And Airplane

This section explains my implementation of rigid body and the `airplane` class with some physics background.

3.5.1 Basics

Given $x(t)$ the position of an object at time t, the velocity is defined as the first derivative of this $v(t) = \dot{x} = \frac{dx}{dt}$ and the acceleration is the derivative of velocity so, $a(t) = \dot{v} = \frac{d^2x}{dt^2}$. We can get position by integrating acceleration as follows: $x = x_0 + v_0 t + \frac{a}{2}t^2$. In the code, the rigid body stores the position and the velocity of the body and the updates are performed using Euler integration:

$$v_{new} = v_{old} + a\Delta t \tag{3.3}$$

$$x_{new} = x_{old} + v\Delta t \tag{3.4}$$

But the rigid body has infinite points, which position do we store?

3.5.2 Center Of Mass

Due to Newton's third law which states that: every action has an opposite reaction, we know that the momentum in a closed system is conserved so the force on a object is equal to the external force. If the object is composed of tiny masses m_i at positions x_i we can write this external force as:

$$F = \sum m_i \frac{d^2 x_i}{dt^2}$$

$$F = \frac{d^2(\sum m_i x_i)}{dt^2}$$

If M is the total mass then:

$$F = M\frac{d^2(\sum \frac{m_i x_i}{M})}{dt^2}$$

$$F = M\frac{d^2 R_{CM}}{dt^2}$$

Where $R_{CM} = \sum \frac{m_i x_i}{M}$ is the center of mass of the rigid body. Since the external force only acts on this point we only need to store this.

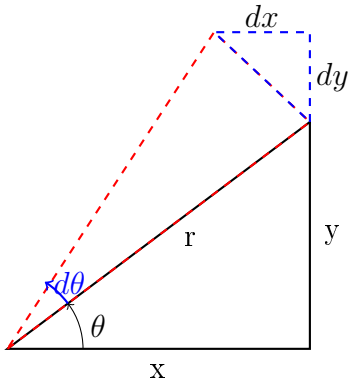
3.5.3 Linear Force

When a force F is applied, the acceleration of the rigid body can be calculated as

$$a = F/m$$

This acceleration is then used with equations 3.3 and 3.4 to update the state of the rigid body. Technically, this should be thought of as an impulse which is force applied for time Δt.

3.5.4 Rotational Motion and Torque



To understand rotational motion take the figure above and imagine the particle is at coordinates (x, y) and is rotated by angle $d\theta$ because of some force. From the picture above it can be seen that $dx = -rd\theta \sin\theta = -rd\theta\frac{y}{r} = -yd\theta$. Similarly, $dy = xd\theta$. From this we get:

$$v_x = \frac{dx}{dt} = -y\frac{d\theta}{dt} = -y\omega$$

$$v_y = \frac{dy}{dt} = x\frac{d\theta}{dt} = x\omega$$

Where $\omega = \frac{d\theta}{dt}$ is called the angular velocity. Since $v = \sqrt{v_x^2 + v_y^2}$ is the tangential velocity, we get:

$$v = \sqrt{y^2\omega^2 + x^2\omega^2} = r\omega$$

.

Since work is defined as Force times the displacement, torque is derived from this. Let's look at what the work is in the above equations:

$$\begin{aligned} W &= F_x dx + F_y dy \\ &= F_x(-yd\theta) + F_y(xd\theta) \\ &= (xF_y - yF_x)d\theta \end{aligned}$$

Work is torque times how much the object rotated.

$$\tau = xF_y - yF_x$$

It can be calculated that this is the derivative of $L = xP_y - yP_x$ which is defined as the angular momentum (where P is the linear momentum). The magnitude of L can be calculated as follows: $L = mv_\perp r = mr^2 w$. Same as the linear momentum, the

term mr^2 can be thought of as the hardness to rotate an object, and this is defined as the moment of inertia: $I = mr^2$.

All equations mentioned above have 3D analogues which are more useful to us. Angular velocity can be imagined as a vector that is perpendicular to the plane of rotation.

Torque in 3D is given as: $\tau = r \times F$. The tangential velocity is given as $v = \omega \times r$. The angular momentum is given as: $L = r \times p$ .These can be imagined using the right hand rule for cross products. For a more thorough physics derivation, consult an introductory physics book.

The other more useful representation for L and τ can be derived as follows:

$$
\begin{aligned}
L &= r \times p \\
&= r \times mv \\
&= mr \times v \\
&= mr \times (\omega \times r) \\
&= mr \times \begin{bmatrix} w_y r_z - w_z r_y \\ w_z r_x - w_x r_z \\ w_x r_y - w_y r_x \end{bmatrix} \\
&= m \begin{bmatrix} r_y w_x r_y - r_y w_y r_x - r_z w_z r_x + r_z w_x r_z \\ r_z w_y r_z - r_z w_z r_y - r_x w_x r_y + r_x w_y r_x \\ r_x w_z r_x - r_x w_x r_z - r_y w_y r_z + r_y w_z r_y \end{bmatrix} \\
&= m \begin{bmatrix} (r_y^2 + r_z^2) w_x - r_y w_y r_x - r_z w_z r_x \\ -r_x w_x r_y + (r_z^2 + r_x^2) w_y - r_z w_z r_y \\ -r_x w_x r_z - r_y w_y r_z + (r_x^2 + r_y^2) w_z \end{bmatrix} \\
&= m \begin{bmatrix} (r_y^2 + r_z^2) & -r_y r_x & -r_z r_x \\ -r_x r_y & (r_z^2 + r_x^2) & -r_z r_y \\ -r_x r_z & -r_y r_z & (r_x^2 + r_y^2) \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} \\
&= I\omega
\end{aligned}
$$

Where I is called the inertia tensor. The off diagonal elements are non-intuitive but since this matrix is symmetric, we can diagonalize it so in the eigenbasis the inertia tensor is just a diagonal matrix.

Similarly, torque can be written as [2]:

$$\tau = I\alpha$$

Where α is the angular acceleration. This equation is the rotational analogue of $F = ma$.

To proceed with rotational updates, we first need to decide how to represent orientation in 3D space.

3.5.5 Orientation in 3D and Quaternions

There are generally three common ways to represent the orientation of a rigid body in 3D. The most well-known method is Euler angles, represented as (α, β, γ), corresponding to rotations around the x, y, and z axes. However, this approach is prone to Gimbal lock. Depending on the order in which you apply the rotations, certain configurations can lead to the loss of a degree of freedom. More insights on this can be found in this YouTube video [17].

Another method is to represent orientation using a rotation matrix R. However, during numerical integration, it's common for the matrix R to lose its orthonormality, requiring periodic re-orthogonalization. Additionally, it requires storing 9 values, which is relatively inefficient.

The most common way of storing orientations in modern physics engines is with quaternions. Quaternions are generalized versions of complex numbers. A quaternion q is represented as:

$$q = a + bi + cj + dk$$

where a is called the real part, and b, c, and d are the imaginary components. If you try really hard, you can imagine this as a unit vector in 4 dimensions.

A video from 3Blue1Brown [18] explains how to visualize quaternions in 3D using stereographic projection. Another video [19] explains how they are used for rotations. However, the most rigorous and understandable derivation I could find was in Oleg Viro's lecture notes [20].

[2] This is not the full equation, ideally we should use Euler's equation to get $\tau = \frac{DL}{Dt} = \frac{d(L)}{dt} + \omega \times L = I\alpha + \omega \times I\omega$. However, this is more involved and for the sake of simplicity I decided to skip the second term by assuming the aircraft is a cube.

The two representations of quaternions useful for us at the moment are as follows: If $q = \cos\frac{\theta}{2} + u\sin\frac{\theta}{2}$, where $u \in \mathbb{R}^3$ is a unit vector, then the mapping $\mathbb{R}^3 \to \mathbb{R}^3 : p \to qp\bar{q}$ describes a rotation of p around the axis u by angle θ.

The other interpretation comes from [19], where we take a unit quaternion $q = w + xi + yj + zk$ and interpret $w = 1.0$ as no rotation, $x = 1.0$ as a counterclockwise rotation around the x-axis, $x = -1.0$ as a clockwise rotation around the x-axis, and so on. This interpretation is especially useful in our context, since we use quaternions directly to represent orientation. Remember, these are unit quaternions—so as we adjust x to go from 1.0 to -1.0, the other values must also change accordingly to preserve unit length:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1$$

Since quaternions are vectors in 4D, using them avoids Gimbal lock entirely. They also make interpolating between two orientations much simpler through SLERP[3]. For all these reasons, I chose to use quaternions for rigid body orientation—just like most modern physics engines do.

3.5.6 Update equations for rotational motion

When a torque τ is applied, we can get the angular acceleration as:

$$\alpha = I^{-1}\tau$$

However, this is not quite right, the inertia tensor is defined in the original basis vectors and not the rotated basis vectors (the orientation) of the body. If R is the current orientation of the body then the correct equation is:

$$\alpha = RI^{-1}R^{-1}\tau$$

Which can be interpreted in the following way: transform torque to the original coordinate system, apply inverse inertia tensor transformation, and transform it back to the rotated coordinate system.

If q is the current orientation of the body given in quaternions then the updates are done as follows:

[3]Spehrical Linear Interpolation (SLERP) can actually be motivated from complex numbers. If you imagine representing 2D angles as complex numbers, and think about how you would interpolate between two angles, this naturally leads to an explanation for interpolating quaternions. However, this is not directly relevant here, so I decided not to go into it.

$$\omega_{new} = \omega_{old} + \alpha\Delta t \tag{3.5}$$

$$q_{new} = q_{old} + \frac{\Delta t}{2}\dot{q}_{old} \cdot q_{old} \tag{3.6}$$

Where the "." represents quaternion multiplication and,

$$\dot{q} = 0 + \dot{q}_x i + \dot{q}_y j + \dot{q}_z k$$

$$\dot{q} = 0 + \omega_x i + \omega_y j + \omega_z k$$

In other words, $\dot{q}$ is the ω vector represented as a quaternion.

A derivation of this fact: $\frac{dq(t)}{dt} = \frac{1}{2}\omega(t)q(t)$ can be found in David H. Eberly's Game Physics book [21].

In practice, q_{new} should also be normalized as it must be a unit quaternion.

3.5.7 The `RigidBody` class

The `RigidBody` class in the code encomposses the functionality described above. The methods `applyTorque`, `applyForce` behave as you would expect. However, this class also provides `applyRelativeTorque`, `applyRelativeForce` methods which apply the provided force relative to the current orientation. This is supposed to be used with other classes as this by itself has no physical meaning and it represents an abstract body. To be able to draw an object it must inherit from the `Model` (or `Mesh`) class and to give it physics properties it must inherit from `RigidBody` class. Nevertheless, a simple example usage is given below:

```
1  #include "rigid_body.h"
2  #include <glm/glm.hpp>
3  #include <glm/gtc/matrix_transform.hpp>
4  #include <glm/gtc/type_ptr.hpp>
5  #define GLM_ENABLE_EXPERIMENTAL
6  #include <glm/gtx/quaternion.hpp>
7
8  int main(){
9      RigidBody obj{/*mass*/ 1.0f, /*pos*/ glm::vec3(0.0f),
10                   /*orient*/ glm::quat(1.0f, 0.0f, 0.0f, 0.0f),
11                   /*inertia tensor*/ glm::mat3(1.0f)};
12
```

```
13      obj.applyForce(glm::vec3(2.0f, 0.0f, 0.0f));
14      obj.applyRelativeTorque(glm::vec3(1.0f, 0.0f, 0.0f));
15
16      obj.update(0.1f);
17  }
```

Code 3.17: Rigid Body example

3.5.8 Airplane

Normally, when modeling aircraft physics one needs to take into account the lift, drag, and other related forces. For instance, An airplane takes off by generating a pressure difference near the wings, it tilts in a similar fashion with the pressure difference being different for both the wings. However, accurately modeling such phenomena is beyond the scope of this thesis. Instead, I use simplified approximations that strike a balance between physical plausibility and computational efficiency.

3.5.9 Estimation of Forces

Thrust Thrust is modeled as a direct force along the aircraft's longitudinal axis, controlled by the throttle input:

$$\vec{F}_{thrust} = \text{throttle} \times \text{maxThrottle} \times \vec{d}_{forward} \tag{3.7}$$

Where $\vec{d}_{forward}$ is the forward direction vector of the aircraft, and throttle is a normalized input value indicated if the forward key was pressed or not.

Control Surface Forces Rather than explicitly modeling airflow over control surfaces, the implementation directly maps control inputs to torques:

$$\vec{\tau}_{pitch} = \text{pitchInput} \times \text{pitchSpeed} \times (1 - |\text{currentPitch}|) \tag{3.8}$$

$$\vec{\tau}_{roll} = \text{rollInput} \times \text{rollSpeed} \tag{3.9}$$

$$\vec{\tau}_{yaw} = \text{yawInput} \times \text{yawSpeed} \tag{3.10}$$

The pitch control includes a factor of $(1 - |\text{currentPitch}|)$ which reduces effectiveness at extreme pitch angles, simulating control surface saturation.

Stabilizing Effects To simulate natural stability without implementing a full aerodynamic model, corrective torques are applied based on the current orientation:

$$\vec{\tau}_{stabilize\_roll} = -\text{currentRoll} \times \text{stabilizingFactor} \tag{3.11}$$

$$\vec{\tau}_{stabilize\_pitch} = -\text{currentPitch} \times \text{stabilizingFactor} \tag{3.12}$$

These torques tend to restore the aircraft to level flight when no control inputs are applied, approximating the static stability of typical aircraft designs.

Cross-Coupling Effects To simulate how real aircraft movements are coupled, additional effects are approximated:

$$\vec{\tau}_{induced\_yaw} = \text{currentRoll} \times \text{inducedYawFactor} \tag{3.13}$$

$$\vec{\tau}_{induced\_pitch} = -|\text{roll}| \times \text{inducedPitchFactor} \tag{3.14}$$

The induced yaw effect simulates the natural tendency of aircraft to yaw in the direction of roll, caused by differential drag on the wings. The induced pitch effect approximates the slight nose-down tendency when banking.

Practical Implementation These estimations, while simplified, create a flight model that captures the essential behavior of aircraft without the computational overhead of true aerodynamic simulation. The result is an intuitive flying experience that maintains physical plausibility while being easy to implement and tune.

```
1 // Apply thrust
2 applyRelativeForce(glm::vec3(0.0f, 0.0f, -1.0f) * throttle *
     maxThrottle);
3
4 // Apply combined control torques
5 applyRelativeTorque(glm::vec3(
6     (pitch*(1-std::fabs(currentPitch)) + stabilizingPitch +
        inducedPitch) * pitchSpeed,
7     (yaw + inducedYaw) * yawSpeed,
```

```
8     (roll*rollFactor+stabilizingRoll*stabilizingRollFactor) *
          rollSpeed
9 ));
```

Code 3.18: Force application in update method

The simplicity of this approach allows for easy adjustment of flight characteristics through the configuration parameters, making it straightforward to tune the model for different aircraft types or desired handling qualities.

3.6 Testing

3.6.1 Building with CMake

Building with instructions specified in Section 3.1 does not automatically build tests too. The testing component was seperated for a faster build time.

The instructions specified in Section 3.1 must be followed as the first step for getting the dependencies then the project with tests can be built with CMake either by running the `build_with_tests.ps1` script (if your vckpg is located in "C:/vckpg") or by doing the following in the shell:

```
1 mkdir build
2 cd build
3 cmake .. -G "Visual Studio 17 2022" -A x64 -DCMAKE_TOOLCHAIN_FILE=%
    PATH_TO_VCPKG%/scripts/buildsystems/vcpkg.cmake -DBUILD_TESTS=
    TRUE
```

Code 3.19: Building with CMake

In other words, you can build tests by setting the `BUILD_TESTS` flag to true in CMake.

3.6.2 Running the tests

The tests can be run on Visual Studio by navigatings to `Test`>`Test Explorer` and clicking on **Run All Tests In View**. This first builds the code and then discovers the tests, if it does not automatically run the tests navigate to `Test` and click on **Run All tests**. The result should look like something show in the figure 3.27 below.

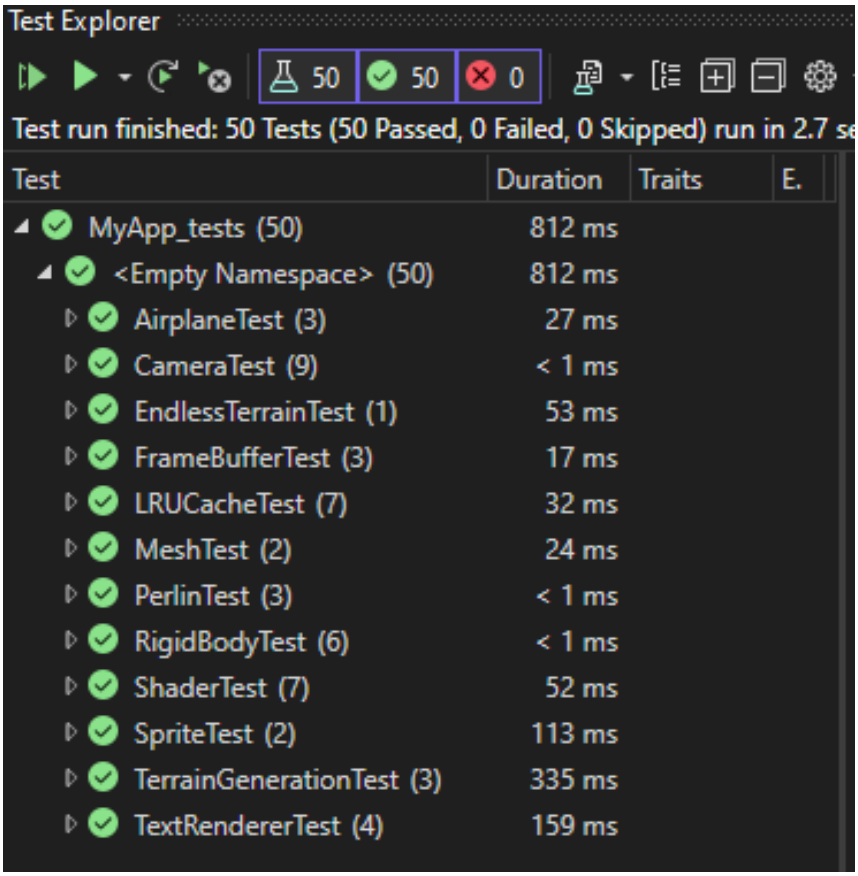


Figure 3.27: Running the tests in Visual Studio

3.6.3 Testing Plan

I decided to use `Google Test (gtest)` as the testing library because of its wide support and easy integration. It's available through vckpg and is installed through the `install_dependencies.ps1` script.

The tests are a mixture of unit and integration tests. The components that do not depend on anything else such as LRUCache, PerlinNoise, RigidBody are tested in the style of unit tests. The other components that depend on OpenGL context to be initialized could have followed a similar style, however, for that a lot of classes would have needed be mocked which would have required an implementation of interfaces causing a change to the design of the project. Instead, these tests are done as a hybrid of unit and integration tests and as I'll explain below for the tests that require an OpenGL context, an empty context is built.

3.6.4 Unit Tests

LRUcache

LRUCache is an independent component and the following tests were done on it:

- Basic insertion and retrieval functionality
- Proper implementation of the LRU eviction policy
- Update of access order when elements are accessed
- Value overwriting behavior
- Size tracking accuracy
- Consistent behavior under repeated access patterns
- Performance under high-volume operations

Perlin Noise

The following tests were done for perlin noise:

- Consistency: Identical inputs should produce identical outputs
- Distinction: Different inputs should produce different outputs
- FBM correctness: The multi-octave implementation should match the mathematical definition

Rigid Body

Following tests were done for rigid body:

- Linear movement in response to forces
- Damping effects on velocity
- Rotation in response to torque
- Combined movement from off-center forces
- Local vs. global coordinate force application
- Orientation normalization under continuous rotation

Camera

The following tests were implemented for the `Camera` component:

- Initialization: Verifies that a new camera instance is oriented toward the negative Z-axis by default.
- Yaw rotation: Confirms that changes in yaw correctly update the front direction vector, including wraparound behavior beyond 360°.
- Pitch clamping: Ensures that pitch values are clamped within a safe range to prevent gimbal lock or distortion.
- Movement handling: Tests the camera's response to directional movement commands, updating its position accordingly.
- Mouse input: Checks that mouse movement results in correct and scaled yaw/pitch adjustments, respecting sensitivity settings.
- View matrix generation: Validates that the generated view matrix is not an identity matrix and reflects the camera's orientation and position.
- Directional consistency: Ensures the front vector used for rendering aligns with the forward direction inferred from the view matrix.

3.6.5 integration Tests

The ability to set up an empty context is defined in `"tests/include/empty_context.h"`. The class `EmptyContext` initializes an empty OpenGL context so that the OpenGL functions do not break. Note that these tests will use extra resources. The following code snippet shows how a class MyTest which wants an empty context for its tests can benefit from the EmptyContext.

```
1 #include <gtest/gtest.h>
2 #include "empty_context.h"
3
4 class MyTest : public EmptyContext {}
5
6 TEST_F(MyTest, TestSomething)
7 {
```

```
8     //OPENGL CALLS WORK HERE
9 }
```

Code 3.20: Testing with Empty Context Example

Shader

The following tests were conducted to verify the correctness and stability of the `Shader` class:

- File loading: Verifies that the shader source loader reads the contents of a file correctly.
- Shader compilation failure: Ensures that invalid GLSL code fails to compile, with proper error checking.
- Shader program linking: Tests successful compilation and linking of a shader program, and validates uniform location queries.
- Error handling on missing files: Confirms that constructing a shader with nonexistent files raises a runtime error.
- Geometry shader support: Checks that a program including a geometry shader compiles and links without error.
- Uniform setting: Ensures that the API for setting various uniform types (float, int, bool, vectors) executes without throwing.
- Resource cleanup: Validates that shader program resources are properly released upon destruction.

Airplane

The following integration tests were performed on the `Airplane` class. These tests depend on an OpenGL context and a mock audio subsystem, and validate behavior that spans both simulation and system-level resource handling:

- Initialization defaults: Verifies that an airplane instance starts with expected physical force settings and triggers looping aircraft sound playback.

- Package drop: Simulates a parachute drop event and checks that a packet is created and the appropriate audio cue is played.
- Drop cooldown: Confirms that drop actions respect an internal cooldown timer, preventing multiple packets from being spawned in rapid succession.

Endless Terrain

The terrain generation and chunk system were validated using a mix of functional and integration-style tests. These focus on correctness, consistency across chunk boundaries, and asynchronous data handling:

- Chunk data correctness: Each generated chunk was validated to contain the correct number of height-normal vectors, with heights in $[-1, 1]$ and normalized values mapped to $[0, 1]$.
- HeightMapWrapper integration: Verified that height data is correctly received and flagged as ready after asynchronous terrain generation completes.
- Chunk alignment (horizontal): Ensured seamless terrain transitions between adjacent horizontal chunks by comparing edge heights and normals.
- Chunk alignment (vertical): Similarly validated the vertical edges of adjacent chunks to prevent seams or mismatches in the visual terrain.

FrameBuffer

The following tests were performed for the FrameBuffer class:

- Check FrameBuffer initializes correctly.
- Check FrameBuffer is complete after init.
- Check clearColor works correctly.

Mesh

As explain below in Section 3.6.6 not a lot of concrete tests can be performed on the Mesh class but the following tests were done to check correctness of some functions:

- Check invalid texture type throws error on draw.
- Check move operator works correctly.

Text Renderer

The following tests were done for text renderer, other tests were done by drawing actual text onto the screen.

- Non existant font throws error
- Adding characters to storage doesn't crash and is successful
- Drawing for different modes doesn't crash
- Rendering long text does not crash

Sprite Renderer

Most tests were done by actually drawinig sprites onto the screen, however, there are following tests in the suite for the `Sprite` class:

- `getRotationMatrix` returns the correct matrix
- `getRotationMatrix` returns the correct matrix after size is changed

3.6.6 Other Visual Tests

Components like Mesh, Model, and the Cloud System don't have methods that can be checked viably, correct loading of a Model implicitly verifies the correctness of Mesh so both Model and Mesh were checked by actually loading models into the scene. Mesh, itself, can be tested by generating custom meshes and drawing them on the screen; as mentioned before, `funcs` namespace provides functions for building a mesh for sphere and torus which can be drawn on the screen to see if the Mesh class behaves as expected.

The same goes for the cloud system: most of its core functionality is in shaders which cannot be tested externally and the only way to test is to play the game and see how the clouds look.

3.6.7 Test Results

Some tests mentioned above failed initially which revelead previously unknown bugs in the code; however, after fixing those all tests were successful.

3.7 Further Improvements

Visuals For Terrain

The visuals could be improved significantly. Currently, the terrain colors are based on a simple normal calculation and no textures are used. Mixing multiple textures based on terrain height usually produces better results, but because the color of the clouds already gives the mountains a snowy look, this approach required the least effort to make the environment look presentable. This still needs some tuning and I intend to work on it in the future.

Another nice addition would be adding grass to the mountains, either with the use of billboards or with some modern sophisticated techniques.

Visuals For Clouds

The clouds could look much better if the noise textures used were hand-crafted. However, I am generating the weather texture maps using a combination of Perlin and Simplex noise, which is why the clouds don't really have an artistic look. Another thing I want to experiment with is using the Beer's-Powder effect [10] for calculating the lighting of the clouds — as seen in the resources mentioned, this tends to produce much more detailed looking clouds.

Other improvements could include using blue noise during raymarching to reduce banding effects, and curl noise to introduce turbulence into the shapes of the clouds.

Shadows

Shadows are an integral part of any realistic scene involving lights. However, they come at a performance cost, and because of that I did not experiment with them as part of this thesis. The realism of the current environment could be enhanced a lot by accurately modeling the shadows of the clouds on the terrain.

More Accurate Aircraft Dynamics

As mentioned previously, accurate modeling of an aircraft requires correctly simulating all the forces acting on it. Adding this feature to the game could open up possibilities of extending it into a combat game or a more realistic flight simulator.

Water

The methods for drawing water with accurate reflection and refraction of the terrain were not described in this thesis because the implementation is complex and it was not the main focus. The implementation was mainly inspired by ThinMatrix's tutorial [22]. One thing to note is that the reflection part of the water currently does not include the clouds; adding this would require additional raymarching and would help make the scene look more realistic.

Chapter 4

Conclusion

This project has been a substantial undertaking, combining real-time procedural terrain generation, volumetric cloud rendering, and rigid body physics within an OpenGL framework using C++. One of the most challenging yet rewarding aspects was ensuring the system remained stable and free of memory leaks, which required careful debugging and attention to resource management.

Throughout this process, I gained valuable insights into graphics programming, performance optimization, and system architecture. The experience of integrating multiple complex systems—terrain generation, GPU-based rendering techniques, and physics simulation for the airplane—has been deeply educational and has significantly improved my engineering skills.

I could not have achieved this without the help of the open-source community and the vast number of learning resources freely available online. I would like to express my sincere gratitude to all the contributors of those resources, and I hope this thesis and project may, in turn, serve as a helpful reference for others working on similar topics.

Looking ahead, there are many ways to extend this work: implementing more physically accurate weather effects, refining collision detection, or enhancing terrain level-of-detail for performance. Nonetheless, this project forms a solid foundation for continued exploration in real-time simulation and rendering.

Bibliography

[1] Sebestian Lague. *Coding Adventure: Clouds*. 2019. URL: https://youtu.be/4QOcCGI6xOU.

[2] Inigo Quilez. *clouds*. 2013. URL: https://www.shadertoy.com/view/XslGRr.

[3] Reinder Nijhoff. *Himalayas*. 2018. URL: https://www.shadertoy.com/view/MdGfzh.

[4] Omar Cornut. *Dear ImGui*. 2014–2025. URL: https://github.com/ocornut/imgui.

[5] JoeyDeVries. *Learn OpenGL*. https://www.learnopengl.com. -.

[6] Ken Perlin. "Improving Noise". In: *ACM Transactions on Graphics* 21.3 (2002), pp. 681–682. DOI: 10.1145/566654.566636.

[7] The Taylor Series. *How to turn a few Numbers into Worlds (Fractal Perlin Noise)*. 2022. URL: https://youtu.be/ZsEnnB2wrbI.

[8] Paul Dawkins. *Calculus III – Parametric Surfaces*. 2023. URL: https://tutorial.math.lamar.edu/classes/calciii/parametricsurfaces.aspx.

[9] Barak Shoshany. "A C++17 Thread Pool for High-Performance Scientific Computing". In: *SoftwareX* 26 (2024). arXiv:2105.00613, p. 101687. DOI: 10.1016/j.softx.2024.101687. URL: https://github.com/bshoshany/thread-pool.

[10] Guerrilla Games. *Nubis: Real-time volumetric cloudscapes in a nutshell*. 2025. URL: https://www.guerrilla-games.com/read/nubis-realtime-volumetric-cloudscapes-in-a-nutshell.

[11] Fredrik Haggstrom. "Real-time rendering of volumetric clouds". MA thesis. Umeå universitet, 2018. URL: https://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf.

[12] GameDev.net. *Horizon Zero Dawn Cloud System*. 2015. URL: https://www.gamedev.net/forums/topic/680832-horizonzero-dawn-cloud-system/.

[13] Juraj Páleník. "Real-time rendering of volumetric clouds". MA thesis. Brno: Masaryk University, 2016. URL: https://is.muni.cz/th/d099f/thesis.pdf.

[14] Maxime Heckel. *Real-time dreamy Cloudscapes with Volumetric Raymarching*. 2023. URL: https://blog.maximeheckel.com/posts/real-time-cloudscapes-with-volumetric-raymarching/.

[15] Wolfgang Engel, ed. *GPU Pro 7: Advanced Rendering Techniques*. CRC Press, 2016. ISBN: 9781040073841.

[16] Wikipedia contributors. *Worley noise — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/wiki/Worley_noise.

[17] GuerrillaCG. *Euler (gimbal lock) Explained*. 2009. URL: https://youtu.be/zc8b2Jo7mno.

[18] 3Blue1Brown. *Visualizing quaternions (4d numbers) with stereographic projection*. 2018. URL: https://youtu.be/d4EgbgTm0Bg.

[19] Sutrabla. *Humane Rigging 03 - 3D Bouncy Ball 05 - Quaternion Rotation*. 2012. URL: https://youtu.be/4mXL751ko0w.

[20] Oleg Viro. *Lecture 5: Quaternions*. https://www.math.stonybrook.edu/~oleg/courses/mat150-spr16/lecture-5.pdf. 2016.

[21] David H. Eberly, ed. *Game Physics*. CRC Press, 2010. ISBN: 9780080964072.

[22] ThinMatrix. *OpenGL Water Tutorials*. 2015. URL: https://www.youtube.com/playlist?list=PLRIWtICgwaX23jiqVByUs0bqhnalNTNZh.

List of Figures

List of Tables

List of Codes