Eötvös Loránd University

Faculty of Informatics

Dept. of Software Technology and Methodology

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

Chapter 2

User documentation

To ensure that the program runs as smoothly as I have tested on my machine the recommended hardware is at least a 3050TI Nvidia Graphics card (with min 4GB ram), and at least 8GB of RAM. The game is shipped as a zip file exclusively for windows so all the required files are inside of it and can be played just by running the executable. For other OS, it must be built using the source files.

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

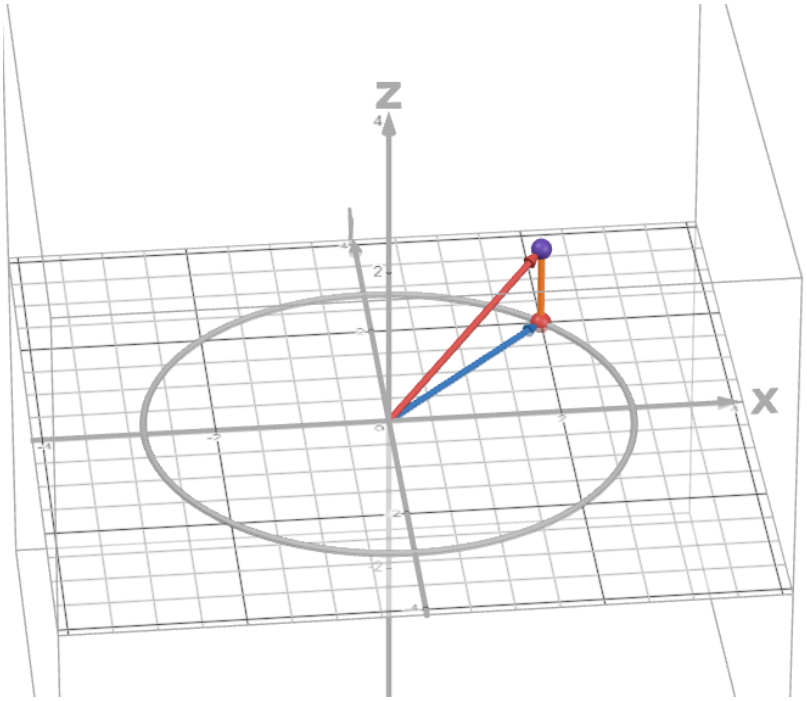

Figure 2.2: Projection of a point in 3d onto the plane

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

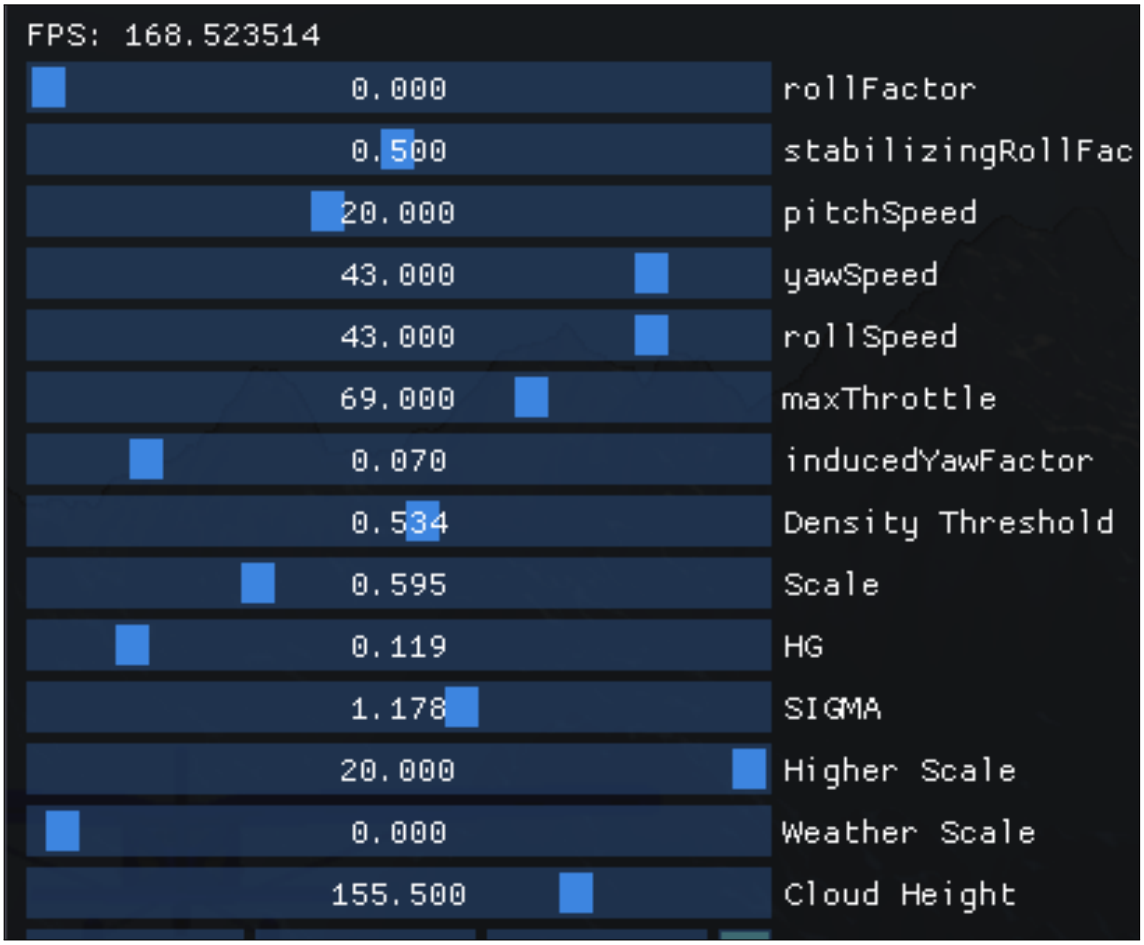

Figure 2.4: The configuration menu

The 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.

Parameter	*Description*
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

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

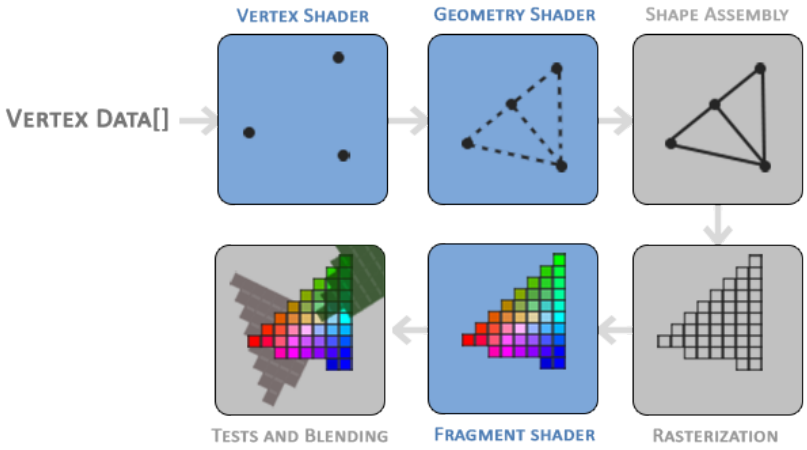

Figure 3.1: The OpenGL Pipeline
[3]

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 like 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 [3]. 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 then 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 [3]. The figure below gives a rough UML of how the mesh class is structured.

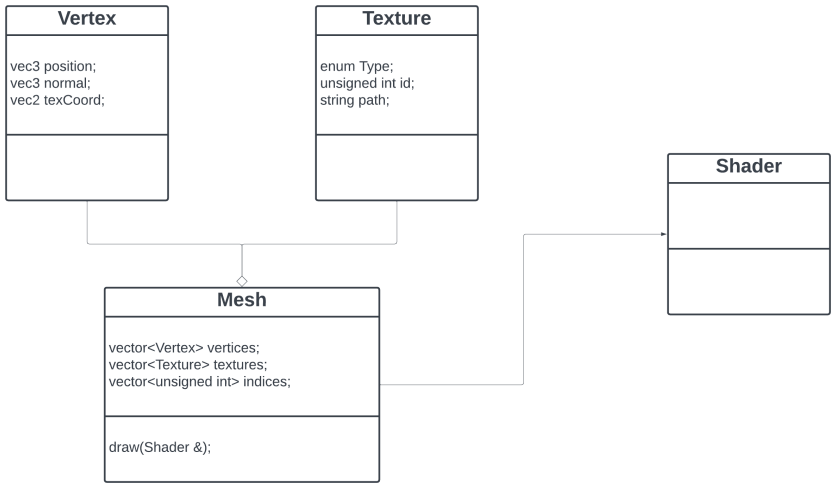

Figure 3.2: Mesh Class
[3]

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 [3], 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.

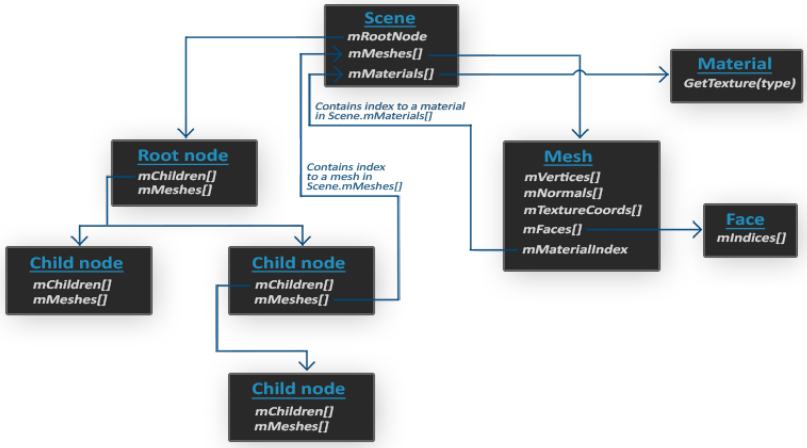

Figure 3.3: Assimp Structure
[3]

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

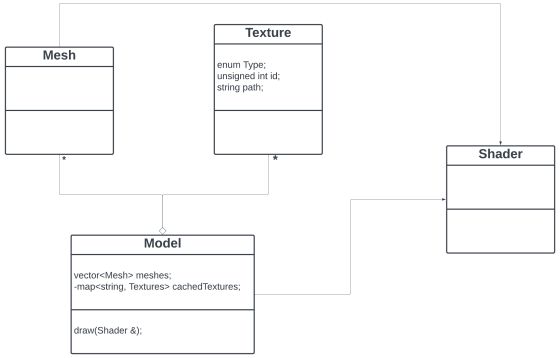

Figure 3.4: Model class UML

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

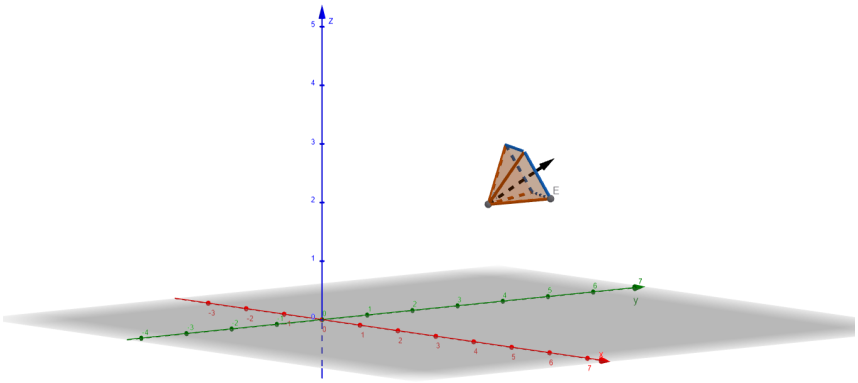

Figure 3.5: Simple Camera example

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>
#include <iostream>

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
    GLuint colorTex = frameBuffer.textureId;
    GLuint depthTex = frameBuffer.depthTextureId;

    // Or, draw the FBO's content directly to the screen
    frameBuffer.draw();
}
```

Code 3.5: Frame Buffer example

3.2.9 Architecture

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

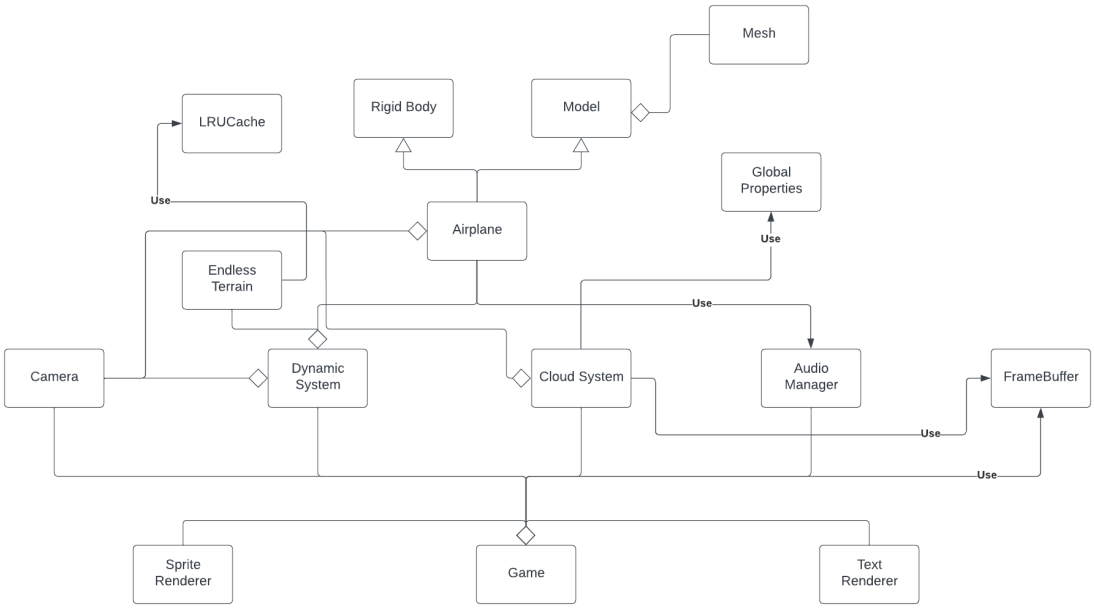

Figure 3.6: Architecture

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** [4], 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
      chunkData.height[i, j] = perlin(x, y)
  return chunkData
```

Code 3.6: 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 [4]. For a more intuitive and visual explanation, see the video referenced in [5].

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.

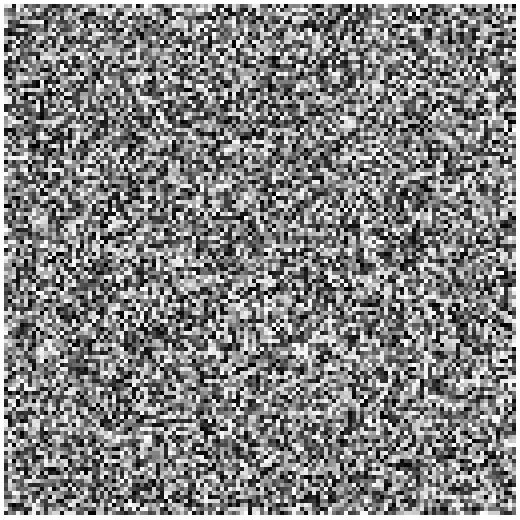

Figure 3.7: White Noise

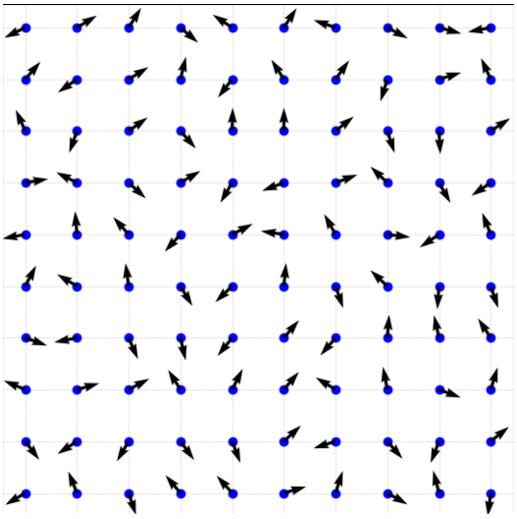

Figure 3.8: Perlin Grid

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 each corner to U:

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

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:

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

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 [6]. 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.

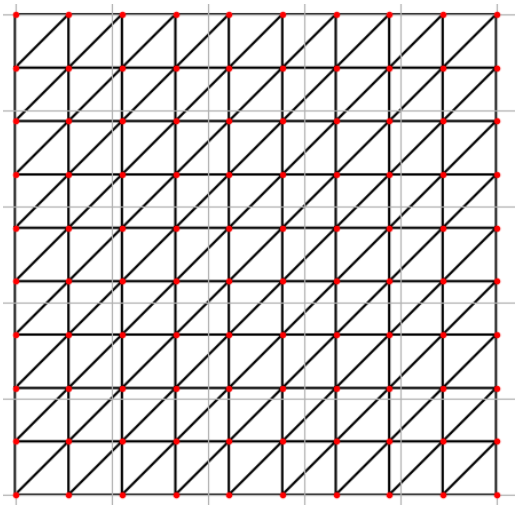

Figure 3.9: Mesh

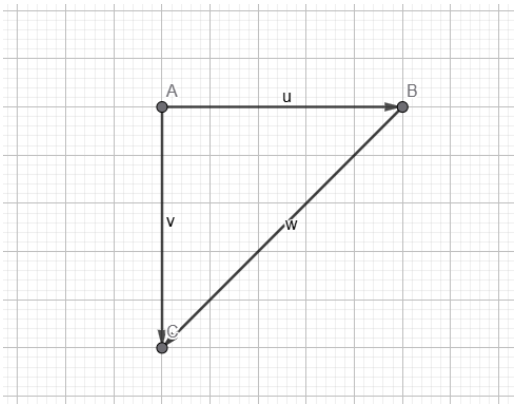

Figure 3.10: Calculating normal of a triangle

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.

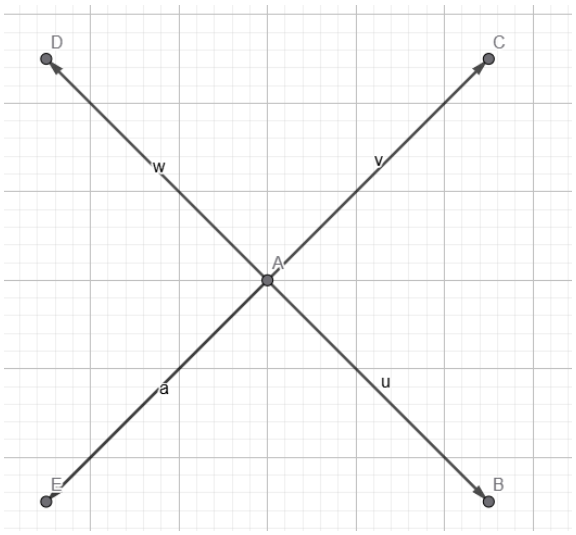

Figure 3.11: Normal Calculation

However, vertex **A** is in multiple triangles. Figure 3.11 shows my approach for calculating the normals. I take the 4 vertices around **A** let's call them $\mathbf{a_0}...\mathbf{a_3}$ in either counterclockwise or clockwise ordering (depending on which way defines the vectors in the up direction). Then the normal vector can be calculated as follows:

$$\vec{n} = \frac{1}{4}\sum_{i=0}^{i=3}(\mathbf{a_i} - \mathbf{A}) \times (\mathbf{a_{(i+1) \bmod 4}} - \mathbf{A})$$

Which is the average of the the normal vectors for each of the 4 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:

```
1 def generateChunkData(size, scale):
2   heightData = Matrix(size + 2, size + 2) // matrix with num rows,
       cols = size + 2
3   for i in range(size + 2):
4     for j in range(size + 2):
5       x = j * scale
```

```
6       y = i * scale
7       heightData[i, j] = noise(x, y)
8
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(4):
16        a = vs[k]
17        b = vs[(k + 1) % 4]
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 3.7: 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 [3]. 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:

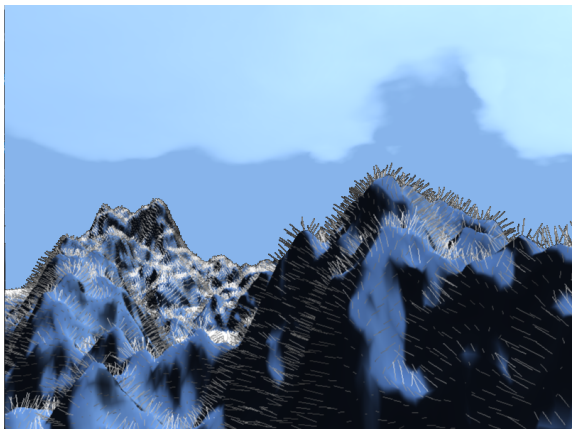

Figure 3.12: Visualizing Terrain Normals

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 3.8: 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 Equations, formulas

Duis suscipit ipsum nec urna blandit, $2 + 2 = 4$ pellentesque vehicula quam fringilla. Vivamus euismod, lectus sit amet euismod viverra, dolor metus consequat sapien, ut hendrerit nisl nulla id nisi. Nam in leo eu quam sollicitudin semper a quis velit.

$$a^2 + b^2 = c^2$$

Phasellus mollis, elit sed convallis feugiat, dolor quam dapibus nibh, suscipit consectetur lacus risus quis sem. Vivamus scelerisque porta odio, vitae euismod dolor accumsan ut.

In mathematica, identitatem Euleri (equation est scriptor vti etiam notum) sit aequalitatem Equation 3.2:

$$e^{i \times \pi} + 1 = 0 \tag{3.2}$$

Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Nullam pulvinar purus at pharetra elementum. Aequationes adsignans aequationis signum:

$$A = \frac{\pi r^2}{2} \tag{3.3}$$

$$= \frac{1}{2}\pi r^2 \tag{3.4}$$

Proin tempor risus a efficitur condimentum. Cras lobortis ligula non sollicitudin euismod. Fusce non pellentesque nibh, non elementum tellus. Omissa numeratione aliquarum aequationum:

$$\begin{aligned} f(u) &= \sum_{j=1}^{n} x_j f(u_j) \\ &= \sum_{j=1}^{n} x_j \sum_{i=1}^{m} a_{ij} v_i \\ &= \sum_{j=1}^{n} \sum_{i=1}^{m} a_{ij} x_j v_i \end{aligned} \tag{3.5}$$

3.4 Source code samples

Nulla sodales purus id mi consequat, eu venenatis odio pharetra. Cras a arcu quam. Suspendisse augue risus, pulvinar a turpis et, commodo aliquet turpis. Nulla aliquam scelerisque mi eget pharetra. Mauris sed posuere elit, ac lobortis metus. Proin lacinia sit amet diam sed auctor. Nam viverra orci id sapien sollicitudin, a aliquam lacus suscipit. Quisque ac tincidunt leo Code 3.9 and 3.10:

```
#include <stdio>

int main()
{
  int c;
  std::cout << "Hello World!" << std::endl;

  std::cout << "Press any key to exit." << std::endl;
  std::cin >> c;

  return 0;
}
```

Code 3.9: Hello World in C++

```
using System;
namespace HelloWorld
{
  class Hello
  {
    static void Main()
    {
      Console.WriteLine("Hello World!");

      Console.WriteLine("Press any key to exit.");
      Console.ReadKey();
    }
  }
}
```

Code 3.10: Hello World in C#

3.4.1 Algorithms

A general Interval Branch and Bound algorithm is shown in Algorithm 1. An appropriate selection rule is applied in Step 3.

Source of example: Acta Cybernetica (this is a hyperlink).

Algorithm 1 A general interval B&B algorithm

Funct IBB(S, f)

1: Set the working list $\mathcal{L}_W := \{S\}$ and the final list $\mathcal{L}_Q := \{\}$
2: **while** ($\mathcal{L}_W \neq \emptyset$) **do**
3: Select an interval X from $\mathcal{L}_W$ ▷ Selection rule
4: Compute $lbf(X)$ ▷ Bounding rule
5: **if** X cannot be eliminated **then** ▷ Elimination rule
6: Divide X into $X^j,\ j = 1, \ldots, p$, subintervals ▷ Division rule
7: **for** $j = 1, \ldots, p$ **do**
8: **if** X^j satisfies the termination criterion **then** ▷ Termination rule
9: Store X^j in $\mathcal{L}_W$
10: **else**
11: Store X^j in $\mathcal{L}_W$
12: **end if**
13: **end for**
14: **end if**
15: **end while**
16: **return** $\mathcal{L}_Q$

Chapter 4

Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In eu egestas mauris. Quisque nisl elit, varius in erat eu, dictum commodo lorem. Sed commodo libero et sem laoreet consectetur. Fusce ligula arcu, vestibulum et sodales vel, venenatis at velit. Aliquam erat volutpat. Proin condimentum accumsan velit id hendrerit. Cras egestas arcu quis felis placerat, ut sodales velit malesuada. Maecenas et turpis eu turpis placerat euismod. Maecenas a urna viverra, scelerisque nibh ut, malesuada ex.

Aliquam suscipit dignissim tempor. Praesent tortor libero, feugiat et tellus porttitor, malesuada eleifend felis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam eleifend imperdiet lorem, sit amet imperdiet metus pellentesque vitae. Donec nec ligula urna. Aliquam bibendum tempor diam, sed lacinia eros dapibus id. Donec sed vehicula turpis. Aliquam hendrerit sed nulla vitae convallis. Etiam libero quam, pharetra ac est nec, sodales placerat augue. Praesent eu consequat purus.

Acknowledgements

In case your thesis received financial support from a project or the university, it is usually required to indicate the proper attribution in the thesis itself. Special thanks can also be expressed towards teachers, fellow students and colleagues who helped you in the process of creating your thesis.

Appendix A

Simulation results

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque facilisis in nibh auctor molestie. Donec porta tortor mauris. Cras in lacus in purus ultricies blandit. Proin dolor erat, pulvinar posuere orci ac, eleifend ultrices libero. Donec elementum et elit a ullamcorper. Nunc tincidunt, lorem et consectetur tincidunt, ante sapien scelerisque neque, eu bibendum felis augue non est. Maecenas nibh arcu, ultrices et libero id, egestas tempus mauris. Etiam iaculis dui nec augue venenatis, fermentum posuere justo congue. Nullam sit amet porttitor sem, at porttitor augue. Proin bibendum justo at ornare efficitur. Donec tempor turpis ligula, vitae viverra felis finibus eu. Curabitur sed libero ac urna condimentum gravida. Donec tincidunt neque sit amet neque luctus auctor vel eget tortor. Integer dignissim, urna ut lobortis volutpat, justo nunc convallis diam, sit amet vulputate erat eros eu velit. Mauris porttitor dictum ante, commodo facilisis ex suscipit sed.

Sed egestas dapibus nisl, vitae fringilla justo. Donec eget condimentum lectus, molestie mattis nunc. Nulla ac faucibus dui. Nullam a congue erat. Ut accumsan sed sapien quis porttitor. Ut pellentesque, est ac posuere pulvinar, tortor mauris fermentum nulla, sit amet fringilla sapien sapien quis velit. Integer accumsan placerat lorem, eu aliquam urna consectetur eget. In ligula orci, dignissim sed consequat ac, porta at metus. Phasellus ipsum tellus, molestie ut lacus tempus, rutrum convallis elit. Suspendisse arcu orci, luctus vitae ultricies quis, bibendum sed elit. Vivamus at sem maximus leo placerat gravida semper vel mi. Etiam hendrerit sed massa ut lacinia. Morbi varius libero odio, sit amet auctor nunc interdum sit amet.

Aenean non mauris accumsan, rutrum nisi non, porttitor enim. Maecenas vel tortor ex. Proin vulputate tellus luctus egestas fermentum. In nec lobortis risus,

sit amet tincidunt purus. Nam id turpis venenatis, vehicula nisl sed, ultricies nibh. Suspendisse in libero nec nisi tempor vestibulum. Integer eu dui congue enim venenatis lobortis. Donec sed elementum nunc. Nulla facilisi. Maecenas cursus id lorem et finibus. Sed fermentum molestie erat, nec tempor lorem facilisis cursus. In vel nulla id orci fringilla facilisis. Cras non bibendum odio, ac vestibulum ex. Donec turpis urna, tincidunt ut mi eu, finibus facilisis lorem. Praesent posuere nisl nec dui accumsan, sed interdum odio malesuada.

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

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

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

[6] Paul Dawkins. *Calculus III – Parametric Surfaces*. Accessed: 2025-04-14. 2023. URL: https://tutorial.math.lamar.edu/classes/calciii/parametricsurfaces.aspx.

List of Figures

List of Tables

List of Algorithms

List of Codes