Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Real-time volumetric cloudscapes for OpenGL

Student

Axel Salaris

Advisor

Randolf Schärfig

Tutor

Achille Peternier

Industrial Partner

Bachelor

Bachelor in Computer Science

Project Code

C10520

Year

2021/2022

Date

September 02, 2022

Abstract

English

One of the hardest challenges of modern computer graphics is the rendering of complex objects in real-time. This thesis focuses on the rendering of clouds with the goal to achieve a realistic and good-looking cloudscape that can be rendered in real-time making use of OpenGL.

This project is heavily inspired by the techniques used for the rendering of the Volumetric cloudscape of Horizon: Zero Dawn, which was shown for the first time at SIGGRAPH 2015.

The use of volumetric billboards, two-dimensional elements that are shown on screen making use of geometry shaders, is the first technique that was implemented to achieve similar results. A more advanced technique to render complex elements like clouds is ray marching, this technique uses rays to detect collisions and adds up, in an iterative way, the color of the current element to build up the final color that will be shown on screen.

The final product is able to render in real-time moving and lit clouds at different times of the day, such as sunrise, noon, sunset, and nighttime in a scene that also features terrain.

Italian

Una delle sfide più difficili della computer grafica moderna è il rendering di oggetti complessi in tempo reale.Questa tesi si concentra sul rendering di nuvole con l'obiettivo di ottenere un paesaggio nuvoloso realistico e di visivamente bello che possa essere renderizzato in tempo reale utilizzando OpenGL.

Questo progetto si ispira fortemente alle tecniche utilizzate per il rendering del paesaggio nuvoloso volumetrico di Horizon: Zero Dawn, mostrato per la prima volta al SIGGRAPH 2015.

L'uso di billboards volumetrici, elementi bidimensionali che vengono mostrati sullo schermo facendo utilizzando di una geometry shader, è la prima tecnica implementata per ottenere risultati simili. Una tecnica più avanzata per il rendering di elementi complessi come le nuvole è il ray marching. Questa tecnica utilizza i raggi per rilevare le collisioni e somma, in modo iterativo, il colore dell'elemento corrente per costruire il colore finale che verrà mostrato sullo schermo.

Il prodotto finale è in grado di renderizzare in tempo reale nuvole in movimento e illuminate nei diversi momenti della giornata, come l'alba, il giorno, il tramonto e la notte in una scena dove è presente anche del terreno.

Contents

Abstract **i**
English . . . i
Italian . . . ii

Assigned Project **vii**
Description . . . vii
Tasks . . . vii
Goals . . . viii

1 Introduction **1**
1.1 Context . . . 1
1.2 Terminology . . . 2
1.3 Goal . . . 2

2 State of The Art **3**
2.1 The Cloudscape of Horizon: Zero Dawn . . . 3
2.1.1 Guerrilla Games . . . 3
2.1.2 Previous Games and Techinques . . . 4
2.1.3 Early Clouds Exploration . . . 4
2.1.4 Modeling . . . 5
2.1.5 Lighting . . . 8
2.2 Other videogame software houses . . . 11
2.3 Billboards . . . 11
2.4 Baseline code . . . 11
2.4.1 Deferred Shading . . . 12

3 Implementation **13**
3.1 Modeling . . . 13
3.1.1 Three-dimensional textures . . . 13
3.1.2 Shaping clouds . . . 15
3.2 Rendering techinques . . . 17

3.2.1 Billboards . . . 17
3.2.2 Blending . . . 18
3.2.3 Ray Marching . . . 20
3.3 Lighting . . . 24
3.3.1 The importance of light in cloudscapes . . . 24
3.3.2 Light probes . . . 25
3.3.3 Visualization of the lighting texture . . . 27
3.3.4 Rotating light source . . . 27
3.4 Skybox . . . 28
3.4.1 Realistic sky . . . 28
3.4.2 Skybox implementation . . . 29
3.5 Graphical User Interface . . . 30
3.5.1 FreeType . . . 30
3.5.2 GUI Implementation . . . 31
3.6 Terrain . . . 31

4 Results **33**
4.1 Cloud looks . . . 33
4.2 Movement and times of the day . . . 34
4.3 World coverage . . . 35
4.4 Performances . . . 35
4.5 Testing . . . 36
4.6 User Interaction . . . 37

5 Conclusions **39**
5.1 Conclusions . . . 39
5.2 Future Work . . . 40

List of Figures

2.1 Killzone Shadow Fall - art directed sky 4
2.2 Clouds classification 5
2.3 Procedural-looking clouds 6
2.4 3D texture 1 7
2.5 3D texture 2 7
2.6 2D texture 1 7
2.7 Height gradients 7
2.8 Coverage signal 7
2.9 Clouds modeling formula 8
2.10 Modeling - no curl distortion 8
2.11 Modeling - curl distortion 8
2.12 Directional scattering 9
2.13 Beer's law 9
2.14 Beer's law - result 9
2.15 Beer's law with Henyey-Greenstein - result 10
2.16 Beer's-Powder law 10
2.17 Beer's-Powder law - effects 10
2.18 Use of billboards - examples 11
2.19 Deferred Shading 12

3.1 Halo: Combat Evolved - Grass texture 14
3.2 Volumetric bonsai 14
3.3 Volumetric human foot 14
3.4 Volumetric 3D Texture 15
3.5 Coverage 2D Texture 16
3.6 Textures combination - Final result 16
3.7 Textured billboard 17
3.8 Visualizing a cone using points 17
3.9 Smoke-like structure - Blending and billboards 18
3.10 OpenGL blending equation 19

3.11 Blending limitations . . . 20
3.12 Traditional ray marching . . . 21
3.13 Sphere tracing . . . 21
3.14 Ray marching volume definition . . . 22
3.15 Outside visualization . . . 22
3.16 Inside visualization . . . 22
3.17 Low noise threshold value . . . 24
3.18 High noise threshold value . . . 24
3.19 Unlit clouds . . . 25
3.20 Lit clouds . . . 25
3.21 Sunrise . . . 25
3.22 Moonlight . . . 25
3.23 Light probes . . . 26
3.24 Lighting visualization . . . 27
3.25 Default light source position . . . 28
3.26 Rotated light source position . . . 28
3.27 Skybox cubemap . . . 29
3.28 Skybox and clouds . . . 29
3.29 Character glyph . . . 30
3.30 GUI close-up . . . 31
3.31 Terrain and clouds . . . 32

4.1 Cloud details . . . 33
4.2 Cloud brighter edges . . . 34
4.3 Loading screen . . . 35
4.4 AMD - artifacts . . . 36
4.5 AMD - loading screen artifacts . . . 37

5.1 Rainbow Six Siege - Dear ImGUI . . . 40

Assigned Project

Description

One of the great challenges of modern real-time computer graphics is the creation and management of very large 3D environments (e.g., city-sized virtual environments, open worlds, etc.). In addition to the complexity related to all the scene elements grounded in the terrain (buildings, roads, trees, and the terrain itself), the sky also plays a very important role, as it is almost constantly within at least a portion of the field of view of the user. It is then not always possible, in terms of quality and final result, to provide low-quality, pre-rendered, static skyboxes to cope with this issue.

This project goes in the direction of exploring a much more modern approach for the simulation of cloudscapes by combining a series of computer graphics techniques adopted in recent videogames that take advantage of the GPU computational power to provide real-time, dynamic 3D volumetric clouds.

Tasks

- Explore the state of the art regarding the topic by starting with the documentation.
- Understanding the basics of an existing OpenGL-based computer graphics engine provided by the teacher that will be used for the implementation of the volumetric cloudscapes.
- Discuss and agree with the teacher on which strategy to adopt for the implementation of the real-time volumetric cloudscapes rendering functionality.
- Carry out the integration of the functionality in the graphics engine, by always keeping the real-time performance requirement in mind.
- Validating the results (in terms of visual quality and performance) with what is provided by the sources used for the implementation and the state of the art.

Goals

- Reading and understanding the state of the art on the topic and proposing an implementation plan using OpenGL and the graphics engine provided by the teacher.
- Implementing real-time 3D volumetric cloudscapes in the graphics engine.
- Write a simple demo that shows the implemented functionality.
- Analyze and compare results against other sources and the state of the art.

Chapter 1

Introduction

This chapter will cover the context of the project, the terminology used throughout the document, and the end goal that has to be reached.

1.1 Context

In modern videogames users search for, other than a good story and fun gameplay, a great visual and sound experience that can fully immerse them in the world they are playing in, and that makes use of leading-edge technology to do so.

That is one of the reasons why modern computer graphics tries to generate breathtaking environments through new techniques meanwhile trying to achieve good results in terms of performance and quality. The creation of huge landscapes, open worlds, and in general the rendering of large 3D environments that players can admire and explore is always more common.

These environments can not always be made up with pre-built and designed assets by 3D artists, it will just take way too much time, that is why videogame software houses started also generating portions of their games' 3D environments procedurally.

The project will cover in particular the implementation of a way to render in real-time a complex cloudscape using advanced computer graphics techniques in OpenGL 4.6.

The Sky is, in fact, one of the parts of a scene that is almost constantly in sight, making it one of the most crucial parts of the scene that, at all times, should look amazing.

1.2 Terminology

Following are the most frequently used terms throughout the document, this will give non-technical readers the possibility to fully understand what is going to be discussed.

- **GPU (Graphics processing unit)**: a specialized electronic circuit designed to manipulate and alter memory to accelerate the rendering process.
- **Ray Marching**: a technique that can be used to visualize virtual environments with the use of rays. For each pixel, a ray is shot, and, by traversing a volume the color of that particular pixel can be determined.
- **Rendering**: the process of generating an image from a mathematical description of a 3D scene, interpreted by algorithms that define the color of each point.
- **Shader**: a set of software instructions used to calculate rendering effects on graphics hardware. A vertex shader defines the operations that have to be executed for each vertex in the 3D scene, meanwhile, a fragment shader defines the operations on a single fragment often also referred to as a pixel.
- **Texture**: an image or some data that is transferred to the GPU and is, generally, applied to an object during the final stages of the rendering pipeline.
- **Pixel**: defines a discrete point in two-dimensional space with its X and Y coordinates.
- **Voxel**: a unit of graphic information that defines a discrete point in three-dimensional space.

1.3 Goal

The main goal of this project is to generate and render in real-time, using a modern OpenGL specification, like OpenGL 4.6, a realistic-looking cloudscape with the use of modern rendering techniques, such as ray marching and the use of billboards.

The implementation of these techniques is heavily inspired by “The real-time volumetric cloudscapes of Horizon Zero Dawn” by Guerrilla Games [1]. The book "OpenGL Superbible" was also a very useful help in resolving the OpenGL-related problems found during the development itself [2].

Other than being beautiful and amazing to look at the cloudscape must also be generated and rendered taking into consideration performance so that they can be rendered in real-time in a reasonable amount of time without therefore slowing the engine.

The final results should then be shown with a demo that demonstrates the procedural generation of a cloudscape that should look as closer to real clouds as possible.

Chapter 2

State of The Art

This chapter will extensively cover the state of the art of real-time volumetric cloudscapes. Starting with an introduction on what major videogame software houses have implemented in the past, passing to brief papers that explain the use of other techniques to achieve similar results, and finally ending on the baseline code this project relies on and started with.

2.1 The Cloudscape of Horizon: Zero Dawn

The following section will give a brief introduction to who Guerrilla Games, the developers of Horizon Zero Dawn, are, and will describe the technologies shown by them at SIGGRAPH 2015, that eventually brought to the cloudscape visible at the launch of the game.

2.1.1 Guerrilla Games

Guerrilla Games is one of the biggest leading game companies based in Europe and is now completely owned by Sony Interactive Entertainment Europe. The company started creating games in the early 2000s, at the time they were well known for the Killzone franchise, games that were completely asset-based.

The company is also well known for always pushing the boundaries of technical and artistic excellence in its games. And this time is no different.

Just after E3 2015 [3], one of the biggest game events in the whole world where Horizon's world premiere [4] took place, Guerrilla Games presented at SIGGRAPH 2015 [5], the premier conference and exhibition on computer graphics and interactive techniques, one of its most impressive technical works ever, the real-time cloudscape of Horizon: Zero Dawn.

2.1.2 Previous Games and Techinques

So far Guerrilla always used assets to represent heavily art-directed skies with amazing results as can be seen in Figure 2.1 (image taken from [1]). They placed elements like clouds using billboards and sky domes, that were previously designed by artists in Photoshop, on which they then could apply lighting.

This was due to the nature of the Killzone franchise, an FPS (First Person Shooter) game where the player movements were restricted to the story and in which the time of the day was static.

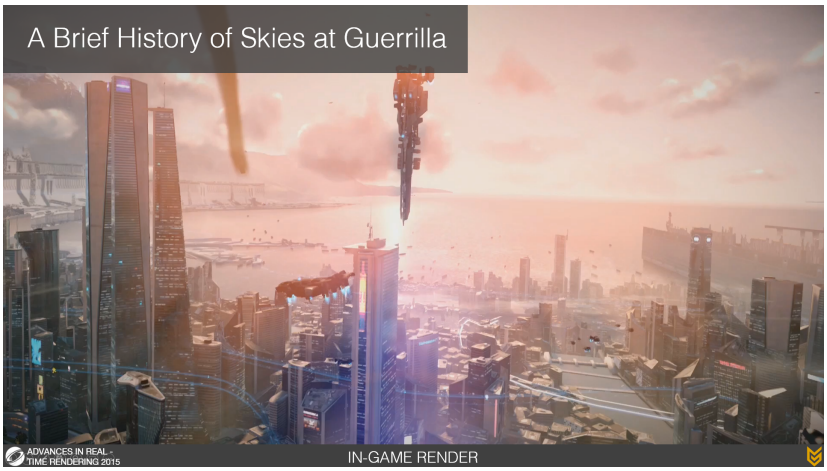

Figure 2.1: Killzone Shadow Fall - art directed sky

2.1.3 Early Clouds Exploration

As previously stated, Horizon is a completely different type of game from those that Guerrilla used to develop. It is a vastly open world where the player can go basically anywhere that he sees, the world is a living world that features a day-time cycle and a weather system, the cloudscape obviously needed to change accordingly. In addition to that skies are a big part of the landscape of Horizon, and are also a very important part of storytelling.

Therefore they started to explore different solutions to solve this problem:

- **Poly Clouds**: treating clouds as part of the landscape, modeling them as polygons and baking the lighting data on them. But this did not work for every type of cloud
- **Billboards**: enhancing the billboard approach to support multiple orientations and times of the day. They succeeded but found that they could not easily re-produce intercloud shadowing
- **Sky Domes**: rendering all of the voxel clouds as one cloud set to produce skydomes that could also blend into the atmosphere over depth

None of those three approaches worked in fact none of them made the clouds evolve over time. Additionally, there was not a good way to make clouds pass overhead, and there was high memory usage and overdraw.

2.1.4 Modeling

After the first failed attempts at building realistic and good-looking clouds Guerrilla decided to opt for voxel clouds, a technique to compose clouds using voxels to give a sense of volume. Although these kinds of techniques are traditionally expensive due to the high amount of texture reads and nested loops, there are many methods to enhance performance and there is convincing work to use noise to model clouds.

To properly model clouds and define where clouds should be drawn, they classified clouds as follows:

- **Strato Clouds**: the lower altitude clouds
- **Alto Clouds**: bandy or puffy clouds above the Strato clouds
- **Cirro Clouds**: big arcing bands and little puffs in the upper atmosphere
- **Cumulonimbus**: the biggest cloud among them, that goes high into the atmosphere

In Figure 2.2 (image taken from [1]) this classification can be seen. A visual representation really helps to understand the different types of clouds before actually implementing them.

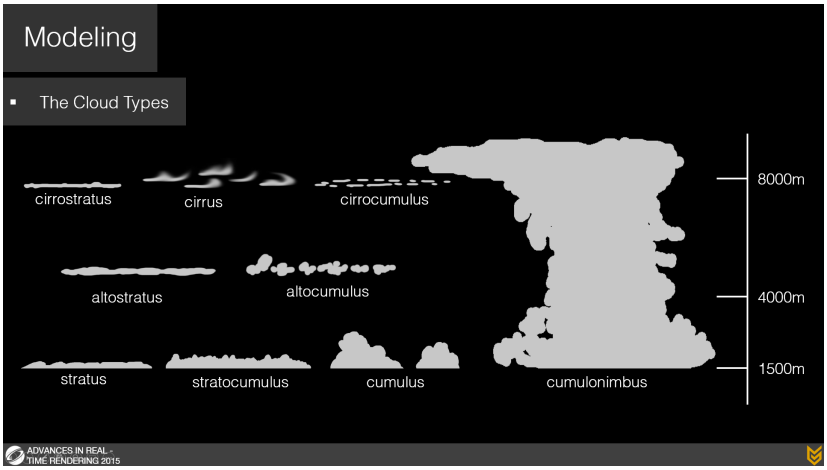

Figure 2.2: Clouds classification

Following the initial clouds classification, Guerrilla modeled the clouds making use of one of the most well-known approaches to model clouds in computer graphics, using Fractal Brownian Noise. This is done by layering Perlin noises of different frequencies and combining them with a gradient to define the cloud's density over height.

This approach is quite easy to implement but presents a major flaw, the resulting clouds are procedural-looking and there is a lack of billow and huge clouds that give a sense of motion. As Figure 2.3 shows (image taken from [1]), the lack of big clouds makes the overall sky look almost motionless.

Figure 2.3: Procedural-looking clouds

To solve this problem, the developers at Guerrilla decided to implement a noise of their own called “Perlin-Worley” noise. As the name can suggest this noise is the combination of Perlin noise with Worley, or cellular, noise. This allowed them to achieve the connectedness of Perlin noise while having some governing billowy shapes.

However, this is not the only noise they used. To furthermore give the clouds a realistic look, they also decided to use and combine the following textures, gradients, and signals:

- **3D Texture 1**: a 128x128x128 resolution RGBA texture. One channel is used to store the Perlin-Worley noise described above, and the other 3 are Layered Worley noises at increasing frequencies. This is used to define the base shape of the clouds
- **3D Texture 2**: a 32x32x32 resolution RGB texture, containing for each channel a Worley noise with increasing frequency. This adds detail to the base cloud shape
- **2D Texture 1**: a 128x128 resolution RGB texture. 3 Curle Noise, this distorts the cloud shapes and adds a sense of turbulence
- **Height Gradients**: a 128 one-dimensional RGB texture. Each channel defines one of the major altitude cloud types (Stratus, Cumulus, and Cumulonimbus)
- **Coverage Signal**: a 128 x 128 resolution RGB texture. Indicates how much cloud coverage is desired at the sample position. Its value goes from zero to 1 and is also used to draw clouds within a 35 km radius, starting at 15km, giving the visible cloudscape and mountains an interesting look. This is the core element of the cloudscape that defines the position of the clouds in space

Figures 2.4, 2.5, 2.6, 2.7, and 2.8 (images taken from [1]) illustrate what is just been described above. Figure 2.6 might be misleading, but the coverage signal is a two-dimensional texture.

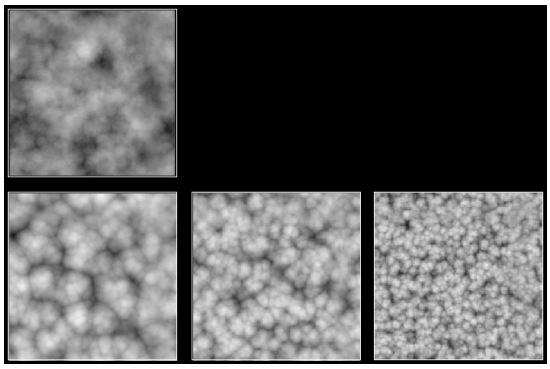

Figure 2.4: 3D texture 1

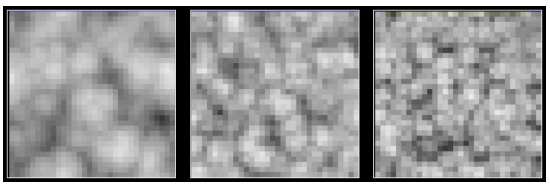

Figure 2.5: 3D texture 2

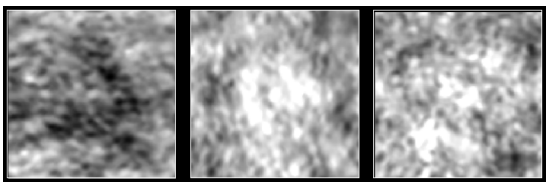

Figure 2.6: 2D texture 1

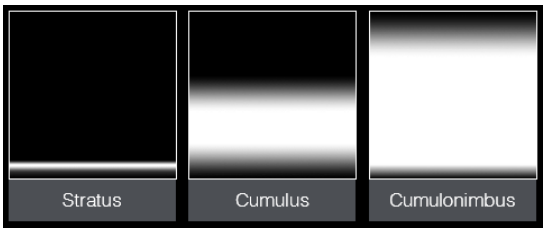

Figure 2.7: Height gradients

Figure 2.8: Coverage signal

To create actual clouds from this series of textures, gradients, and signals Guerrilla decided to use a technique called Ray Marching that consists in shooting a ray for each pixel and, by traversing the 3D volume determining its particular color. The steps are as follows:

- **Step 1**: Sample 3D Texture 1 & Multiply this by the height signal to build low-resolution clouds
- **Step 2**: Erode and multiply the result by a coverage signal (0 to 1) and reduce density at the base of the cloud
- **Step 3**: Sample 3D Texture 2 & erode the base cloud at its edges
- **Step 4**: Distort 3D Texture 2 by 2D curl noise

This can also be represented in the formula form shown in Figure 2.9 (image taken from [1]).

Figure 2.9: Clouds modeling formula

The results are realistic-looking clouds rendered using only different types of noises that completely replace the old asset-based solution Guerrilla used in their previous games. The end result is shown in Figures 2.10 and 2.11 (images taken from [1]).

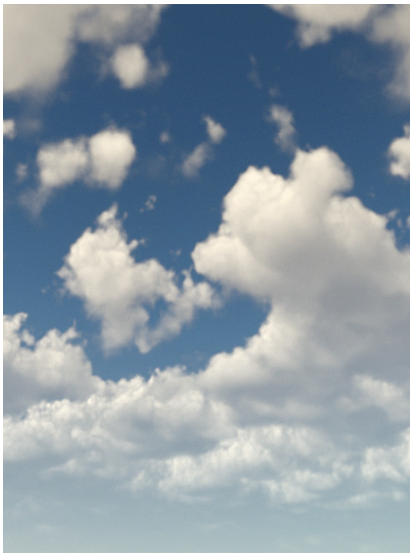

Figure 2.10: Modeling - no curl distortion

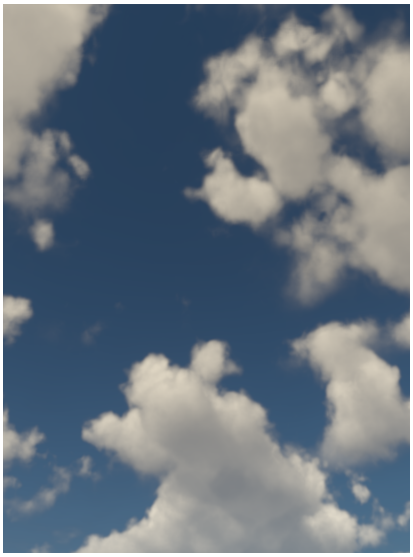

Figure 2.11: Modeling - curl distortion

Guerrilla also mentioned that they managed to animate the clouds. To do so they used the coverage texture, as said before the core element of the cloudscape engine, as it is directly linked with the weather system and the change of its values produces the actual movement of clouds. This reflects the changes in weather in the living world of Horizon.

2.1.5 Lighting

Just as important as modeling, if not even more important, is the way clouds interact with light. Guerrilla wanted to achieve what they considered the three most important lighting effects:

- **Directional scattering**: the luminous quality of clouds
- **Silver lining**: sun effect on clouds when the camera looks towards it
- **Dark edges**: effect on clouds when the camera looks away from the sun

When dealing with directional scattering the possible scenarios that might be encountered are three:

- **Out-scattering**: by the time the light ray finally exits the cloud it could have been out scattered
- **Absorption**: the light ray is completely absorbed by the cloud
- **In-scattering**: the light ray is combined with other light rays

In Figure 2.12 (image taken from [1]) it is shown what scattering inside a cloud could look like in all these scenarios.

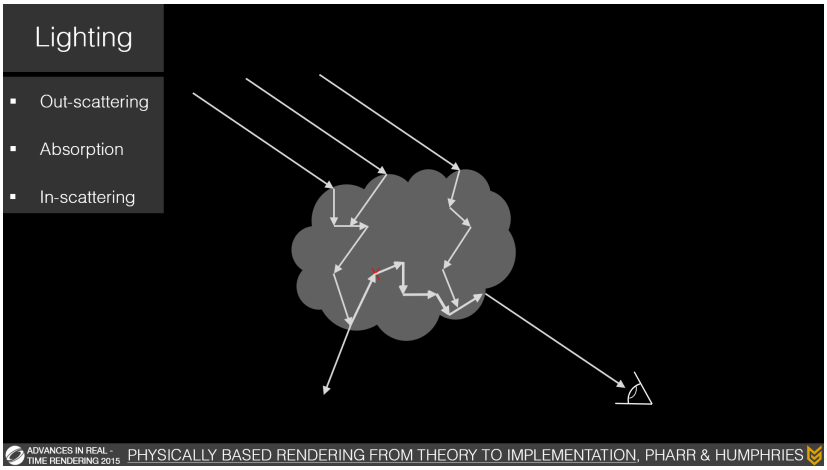

Figure 2.12: Directional scattering

To determine the amount of light at a given point in the cloud, to handle primary scattering, Guerrilla used Beer's Law shown in Figure 2.13 (image taken from [1]), substituting energy for transmittance and depth in the cloud for thickness, the transmittance of light exponentially decreasing over the cloud depth can be seen. The results of its applications are shown in Figure 2.14 (image taken from [1]).

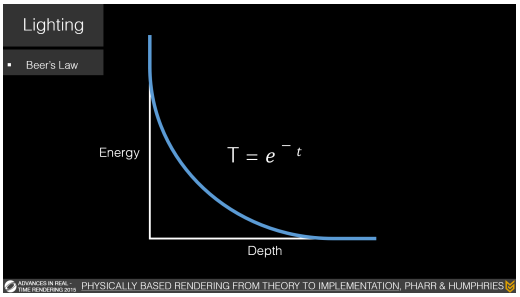

Figure 2.13: Beer's law

Figure 2.14: Beer's law - result

To add the Silver lining effect on the clouds when the camera points towards the sun, Guerrilla combined Beer's Law with the Henyey-Greenstein phase function, of which they do not share any particular information as it is a standard approach. Once this is implemented the result is that the clouds are brighter around the sun, this can be seen in Figure 2.15 (image taken from [1]).

Figure 2.15: Beer's law with Henyey-Greenstein - result

Finally, to mimic the dark edges of clouds as the player looks further away from the sun, the developers at Guerrilla thought to mix the previously used Beer's Law with a Powder function giving birth to what they address as Beer's-Powder Law that can be seen in Figure 2.16 (image taken from [1]).

In Figure 2.17 (image taken from [1]) it is visible how the Beer's Law component handles the primary scattering, the Powder Effect produces the dark edges and their combination produces realistic lighting on the cloud.

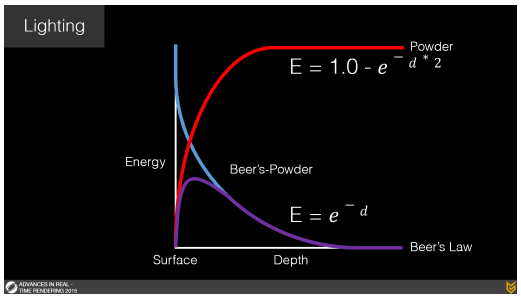

Figure 2.16: Beer's-Powder law

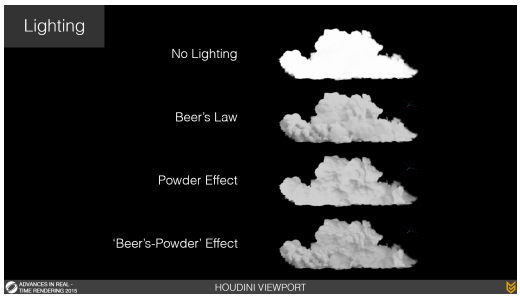

Figure 2.17: Beer's-Powder law - effects

This effect simulates well what goes on inside a cloud, as the points inside a cloud get much more in-scattered light, they are much brighter than the ones outside. And this effect is much more noticeable when the camera looks away from the light source, in this case the sun.

2.2 Other videogame software houses

Other videogame software houses implemented similar techniques such as RockStar Games in Red Dead Redemption 2 however they did not achieve as good results as Guerrilla did and were not as open on their implementation as Guerrilla was. These techniques can nowadays also be found in modern videogame engines such as Unity.

2.3 Billboards

There are many ways to render in real-time semi-transparent and visually complex objects. The approach to do that was previously shown, but it is not the only technique that can be used. Complex shapes can be represented using a series of billboards as thoroughly explained in the 2011 paper Volumetric Billboards [6].

These kinds of techniques are called image-based representations and are often seen as attractive alternatives to regular geometry. This is because with said techniques it is possible to represent geometrically complex objects, including objects made of many small, thin, and possibly disconnected elements that become individually indistinguishable at a distance but still predominantly contribute to the overall shape of the object. Figure 2.18 illustrates some examples of complex objects, which are taken from the above-mentioned paper, rendered in real-time with the use of this technique.

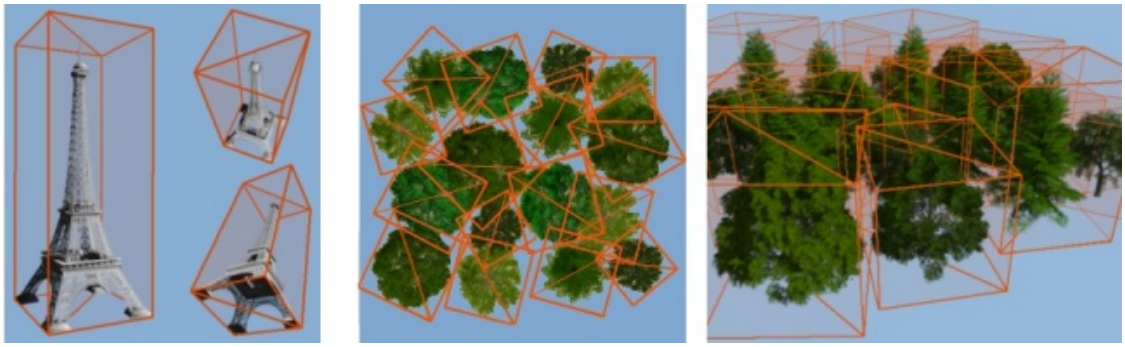

Figure 2.18: Use of billboards - examples

As is visible from these examples billboards could be useful to represent semi-transparent and complex objects such as clouds.

2.4 Baseline code

As previously explained the project base code was given at the start of the project itself by the project's relators, this is due to a multitude of factors.

First of all, it was just impossible in terms of time, to build a fully functional engine and have the time to properly focus on a not-so-simple and conventional task such as the creation of a volumetric cloudscape. In the second place, the relators were already familiar with the code so if something were to be confusing there would not be any sort of problem. And finally, the code itself was already properly tested.

2.4.1 Deferred Shading

The baseline code uses a technique called deferred shading to render the scene, this technique will be described thoroughly here in contrast to the standard forward rendering.

The approach that many use to render a scene is called forward rendering or forward shading. It is a straightforward approach where each object is rendered and lit according to all the light sources in the scene. Doing this for every object in the scene is quite easy but extremely expensive performance-wise as each rendered object has to iterate over each light source for every rendered fragment. On top of that, a lot of fragment shader runs are wasted when multiple objects cover the same screen pixel as their outputs are overwritten.

Deferred shading or deferred rendering in contrast to the conventional forward rendering changes drastically the way objects are rendered. This allows the optimization of a scene with many lights in a way that potentially hundreds, or even thousands, of lights could be rendered with an acceptable framerate.

Deferred shading consists of two passes: in the first pass, called the geometry pass, the scene is rendered once and all kinds of geometrical information are retrieved from the objects that are then stored in a collection of textures called the G-buffer. In a second pass called the lighting pass, the previously stored textures are used in the G-Buffer to compute the scene's lighting for each fragment and render the result in a full-screen quad.

Figure 2.19 illustrates the process of deferred shading, as it can be found at LearnOpenGL [7].

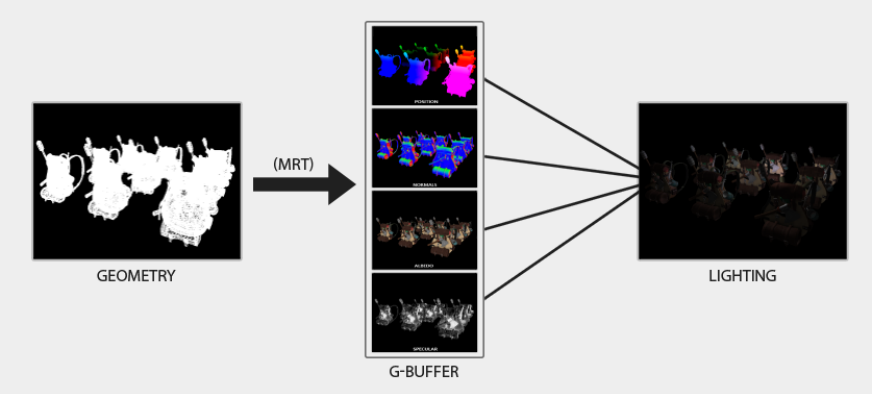

Figure 2.19: Deferred Shading

Chapter 3

Implementation

The following chapter will describe the architecture of this project, in particular the main building blocks that allowed the baseline code to evolve into an OpenGL program that permits the visualization of a volumetric cloudscape taking into consideration the real-time performance requirements.

The project itself can, in fact, be torn down into three main parts. The modeling of the clouds, the rendering techniques that have been implemented, and the lighting of said clouds.

3.1 Modeling

This section will describe the OpenGL data structures necessary to render a volumetric object and the way they are used to achieve realistic and overall good-looking cloudscapes.

3.1.1 Three-dimensional textures

Textures are and have been used in computer graphics applications, videogames, and movies for a long time. These graphical elements are used to give, as their name suggests, a texture to some object, therefore a texture can be wrapped around a 3D object to give it some desired features. But they are also used to give flat surfaces an illusion of depth.

Textures are handy elements to enrich a virtual environment and to give the user of an application that uses said technologies a sense of realism and immersion. A great example of their use is the texture of grass used in old video games like Halo, as shown in Figure 3.1. This particular scene is not present in the original game and is in fact taken from a screenshot of a fan-made mod [8].

Figure 3.1: Halo: Combat Evolved - Grass texture

The above-cited applications of textures, though, applies to two-dimensional textures and will not be of much help to store sufficient data in a meaningful way that could be later on used to render a volumetric object. This can however be achieved with the use of three-dimensional textures.

Three-dimensional textures have three dimensions: width, height, and depth, and maintain all the features of two-dimensional texture. This makes these structures perfect when working with data that is linked in all three dimensions.

The use of three-dimensional textures does not vary much from their two-dimensional alternative but there is not much documentation and examples of their use on the internet. These structures are mostly used to visualize medical data such as bones, blood vessels, and general medical information, but also to visualize complex objects such as trees with many branches and leaves.

In Figure 3.2 and Figure 3.3 a few examples of volumetric rendering in WebGL that uses three-dimensional textures [9] can be seen. All the information needed to render these examples are stored in 3D textures that are later on sampled to give the shown results.

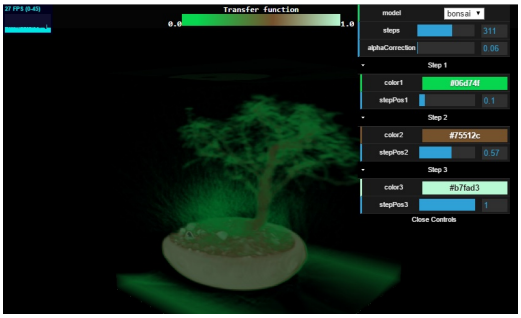

Figure 3.2: Volumetric bonsai

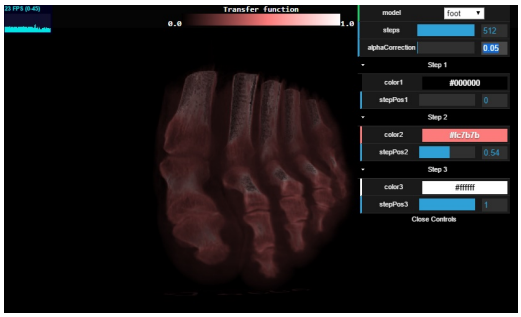

Figure 3.3: Volumetric human foot

3.1.2 Shaping clouds

To build realistic and appealing clouds, noise is the key as it gives a natural look to the cloud. But to achieve something that looks natural, the choice of a good kind of noise is critical, as was previously shown in the state-of-the-art chapter.

At the very start of the application, the necessary textures are generated and loaded in memory. They are subsequently bound to the correct shader program to render them correctly on screen.

This project combines different textures following the Guerrilla Games approach discussed in the state-of-the-art chapter. The generated and used textures are the following:

- **Volumetric 3D Texture**: a 128x128x128 resolution RGBA texture. One channel is used to store tileable Perlin Noise, and the other 3 channels are used to store Worley noises at increasing frequencies.
- **Coverage 2D Texture**: a 128x128 resolution RGBA texture. The first three channels define the different types of clouds, low clouds, mid clouds, and big clouds areas respectively. The fourth and last channel is used to limit the clouds into an inner and outer radius, as the 15km and 35km Guerrilla Games used in Horizon Zero Dawn.
- **Height 1D Texture**: a 128x1 resolution RGBA texture. The first three channels are responsible for the height position of the corresponding three types of clouds previously defined in the coverage texture. The fourth channel is responsible for the definition of a height gradient that decreases the density of the clouds on the bottom.

The volumetric 3D Texture is the heart of the project as it contains the core information to generate consistent noise that gives the desired cloud look. The way this texture is built is crucial, the different Worley Noises need to have substantially different frequencies. If the Worley Noises do not differ much, the cloud will look too regular and the overall chaoticness of real clouds would be lost.

Figure 3.4 shows the 4 different channels of a single slice of the texture.

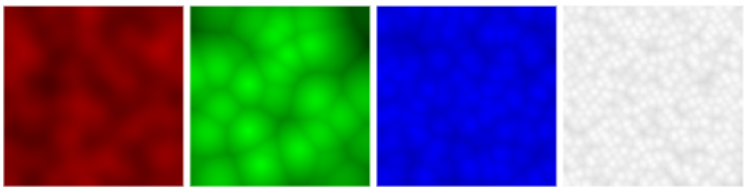

Figure 3.4: Volumetric 3D Texture

The generated textures are visualized using NVIDIA® Nsight™ Graphics, a standalone developer tool that enables debugging, profile, and exporting frames built OpenGL, OpenVR, Vulkan, and many more computer graphics APIs.

The coverage 2D Texture in combination with the previously defined volumetric texture can generate different areas, randomly generated at the start of the application, that define where the different types of clouds should lie. This texture is then modified at runtime with the use of multiple threads to gain performance to simulate the movement of clouds.

Figure 3.5 shows what the different channels of the texture look like, and shows well the areas where the different types of clouds should be positioned at.

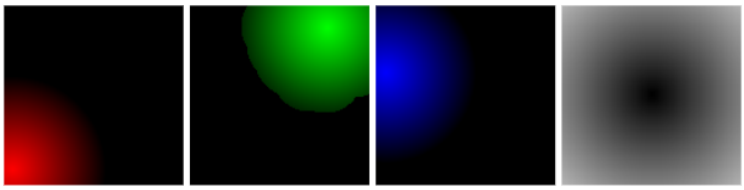

Figure 3.5: Coverage 2D Texture

Now that the areas where clouds should be positioned are defined is necessary to tell where in the y-axis these clouds should be. The 1D Height Texture does exactly this. It contains defined height ranges that in combination with the coverage texture define different types of clouds.

The results of the combination of these textures can be seen in figure 3.6.

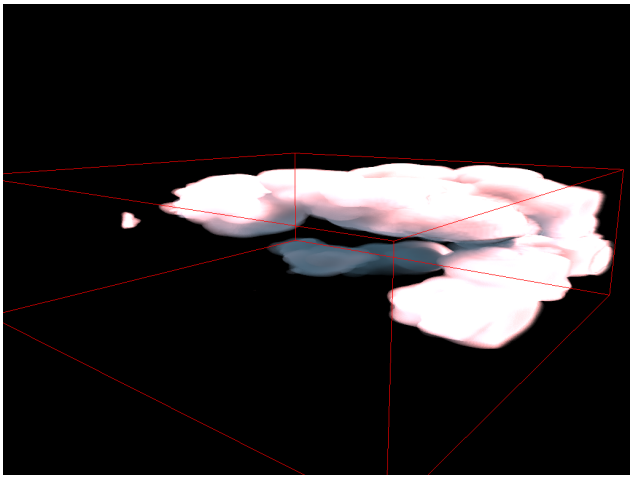

Figure 3.6: Textures combination - Final result

The resulting cloudscape is chaotic as expected. This effect is given by the huge difference in the Worley Noise frequencies. The first Worley Noise Layer has a set number of points, the second one has that number of points multiplied by 10, and the third one by 50. If this difference was not as strong, for instance, if the second layer had two times the overall points of the first layer and the third layer had four times the amount, the generated clouds would look way too regular. thee chaoticness of the cloud would not therefore, be achieved.

3.2 Rendering techinques

This section will describe the different techniques implemented to visualize the 3D texture containing noise and the advantages and disadvantages of each technique.

3.2.1 Billboards

After the generation of the necessary textures, a fundamental of the building of a volumetric cloudscape is finding an effective way to show on screen, or render, the content of the main volumetric 3D texture as it contains all the modeling data for the clouds.

There are a few techniques that enable us to do that, but as 3D textures by themselves were already pretty challenging to understand and correctly implement, and no feedback on screen was so far shown, billboards were used since they were the easiest technique to implement.

As previously thoroughly explained in the state-of-the-art chapter billboards are graphical elements that enable a computer graphics application to render very complex objects without the need for actual geometry, the object is instead composed of little canvases.

To visualize the content of a 3D texture, a specific class called “engine_pipline_billboards” was created in which a Vertex Buffer Object, or VBO, containing a point for each voxel in the texture was used. This allowed us to send the GPU all the necessary data to correctly place a point into a three-dimensional space at the very start of the application.

Points by themselves are not sufficient to show in a meaningful way the content of a texture, that is due to the amount of space taken by the point itself. Billboards are generated from such points, textured as shown in Figure 3.7, and then rendered as a whole. Figure 3.8 shows a cone rendered that way.

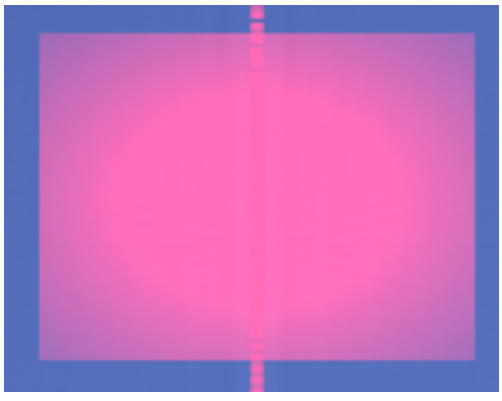

Figure 3.7: Textured billboard

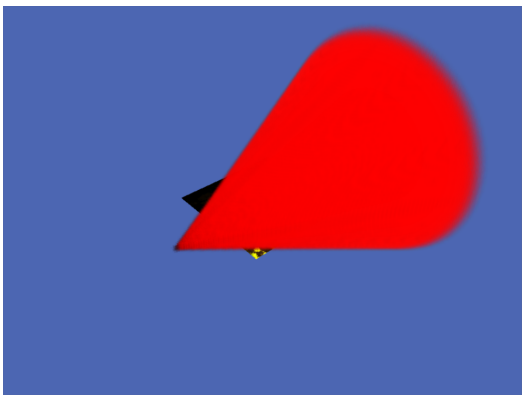

Figure 3.8: Visualizing a cone using points

For this exact reason, the billboard class other than the usual vertex and fragment shaders is also composed of a third shader, the geometry shader.

The geometry shader is responsible for creating billboards, actual quads made of 2 triangles, starting from the point passed from the vertex shader, in clip-space, as their center. The shader gets the current rendered point from the vertex shader and transforms that one point into 4 points that compose the final quad that will be rendered on the screen. Since the geometry shader in this case works in clip-space, the quads will always face the camera direction making the use of this shader really useful.

Each billboard is colored based on the value contained in the 3D texture that is being sampled into the fragment shader, therefore giving the correct effect. As shown previously they are also textured in a way that makes the transition with the background, or perhaps other billboards much smoother.

At this point, an object can be rendered correctly on screen but only the outer layer of the said object can be seen as blending is not active. Clouds are not made of opaque material that blocks the view into them, but they are made of see-through material or water molecules.

3.2.2 Blending

To render semi-transparent objects like clouds and achieve the effect of accumulation of colors as the vision progressively sees inside the cloud itself, additive blending was used.

Additive blending is a type of blending that is used by adding different colors together to eventually build up a final resulting color. In this specific case that allowed the discovery of which parts of the clouds were the densest and which were less dense.

The results of rendering the 3D texture with billboards in combination with additive blending can be seen in Figure 3.9 which shows a smoke-like structure. This works well as multiple points, of different billboards, contribute to the coloring of every pixel in two-dimensional space. The contribution of each point of the volumetric texture can be perfectly seen.

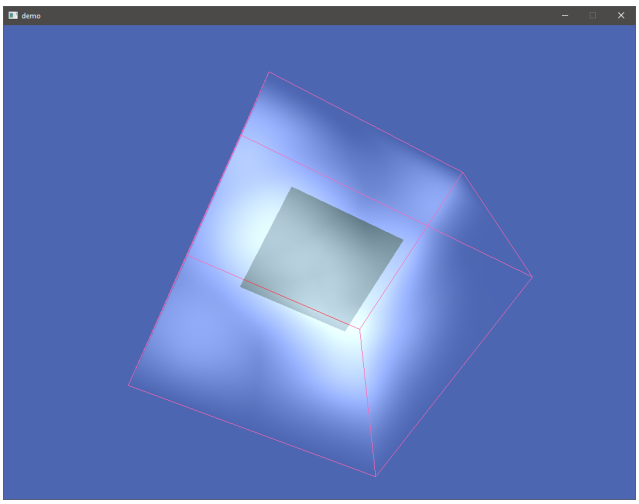

Figure 3.9: Smoke-like structure - Blending and billboards

These results were really useful as the noise with which the 3D texture was previously filled can be at this point seen and modified if needed. This blending effect is achieved using the following OpenGL blending settings:

```
//Disable writing to zBuffer:
glDepthMask(false);

// Enable blending:
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);

// Render stuff:
...

// Disable blending:
glDisable(GL_BLEND);

// Re-enable writing to zBuffer:
glDepthMask(true);
```

Listing 3.1: Blending settings

To understand what these settings mean, the way blending works must first be understood. OpenGL Blending works with the equation shown in Figure 3.10. The full explanation of how this works and how to properly implement blending in OpenGL can be found in the LearnOpenGL blending chapter [10].

$$C_{result} = C_{source} \cdot F_{source} + C_{destination} \cdot F_{destination}$$

- C_{source} = the source color vector. This is the color output of the fragment shader.
- $C_{destination}$ = the destination color vector. This is the color vector that is currently stored in the color buffer.
- F_{source} = the source factor value. Sets the impact of the alpha value on the source color.
- $F_{destination}$ = the destination factor value. Sets the impact of the alpha value on the destination color.

Figure 3.10: OpenGL blending equation

In this case, the source factor is set to be equal to the alpha component of the source color vector. And the destination factor is set to be equal to one. These particular settings are the results of a lot of trial and error to see which gave the best and desired results, to make the content of the texture look like something similar to a cloud.

Blending was useful for a first exploration of the 3D texture but had its limitation. First of all, blending did not allow to stop after a said amount of traveled distance inside the clouds. It did not take into consideration if the maximum amount of light that could reach the user's eyes was reached therefore giving some areas that were extremely bright as Figure 3.11 shows.

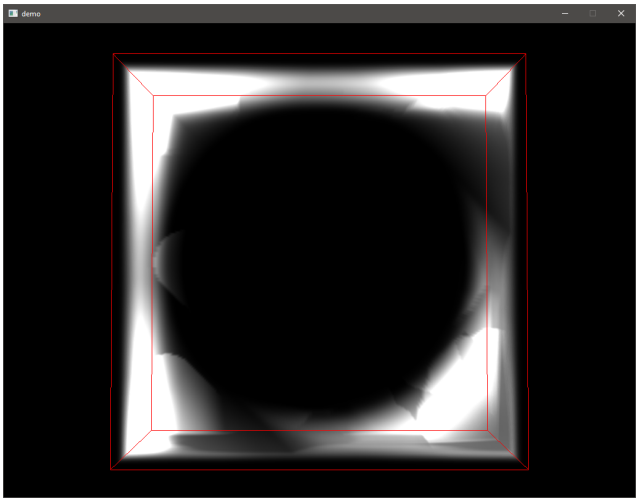

Figure 3.11: Blending limitations

Secondly, the use of billboards and blending had a great impact on performance as for every pixel on the screen blending needed to be computed taking into consideration every visible billboard on that specific pixel. Also, the use of bigger textures meant the use of more and more billboards that eventually drastically reduced the framerate of the application, the engine started to run under 10 FPS, compromising the real-time performance requirement.

A better and more efficient technique to get more accurate results was at this point needed.

3.2.3 Ray Marching

Ray marching is an interesting rendering technique that is based on the concept of shooting rays for each pixel that is currently visible in the scene. The leading idea that lies behind it is that its iterative approach, unlike normal ray tracing where only the intersection between the ray and the object is computed, ray marching samples a volume/model in multiple steps. This is quite useful to render volumetric objects that are not uniform, just like clouds.

Since in this case the first intersection is not known a fixed step size is used. A representation of how ray marching works can be seen in Figure 3.12, as illustrated and explained in a Stack Overflow post regarding this topic [11].

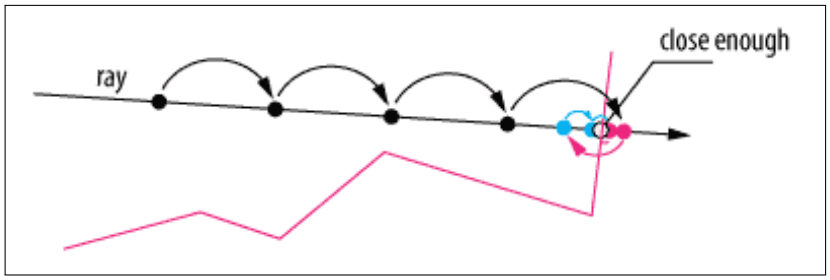

Figure 3.12: Traditional ray marching

A slightly modified version of ray marching that is always more popular among computer graphics enthusiasts is sphere tracing, which can be seen in Figure 3.13 (image taken from [12]). This technique makes use of a signed distance function, or SDF, which, by definition, returns the distance to the closest object in the scene. If this distance is treated as the radius of a sphere centered around the current position, it is possible to know if it is possible to safely move forward along the ray by the radius amount without missing any objects.

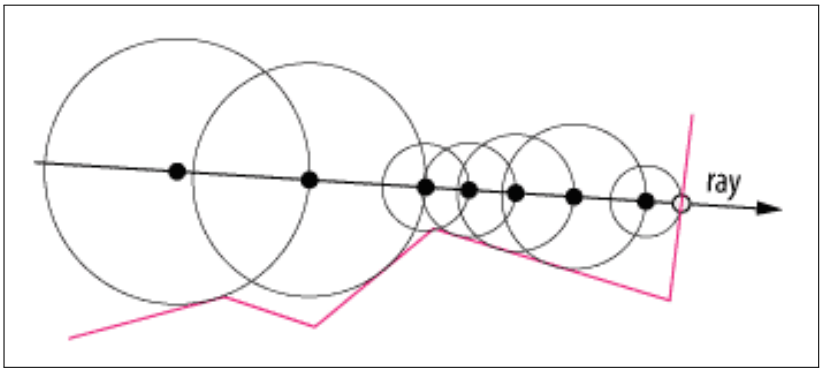

Figure 3.13: Sphere tracing

As appealing as this technique may sound it cannot be used for semi-transparent objects like clouds as there is no way to define a signed distance function. To have a better insight into how this technique is implemented there is a useful article written by Michael Walczyk on his website [12], that explains ray marching and sphere tracing.

The ray marcher implemented in this project makes use of two classes. The first one is the “engine_pipeline_texture3drender” and the second one is the “engine_raymarcher”.

The first class is responsible for the rendering of the whole scene into a texture that will, later on, will be shown on the screen thanks to the deferred rendering logic. No geometry apart from a single full-screen quad, that uses an orthographic projection matrix to always be displayed in front of the screen, is rendered. The ray marcher logic is located in the fragment shader of this class. The shader bounds the 3D Noise texture, the 2D Coverage texture, the 1D Height texture, and the two textures that are generated and modified at each frame into the second class that allows the ray marcher to work properly.

But to do that, the ray marcher needs to define a direction for each ray it shoots, therefore a ray direction for every rendered pixel is needed. This is done by the second class which is responsible for the creation of a cuboid that defines where the clouds will be visualized on screen. This class renders the cuboid two times and outputs the result into two different textures. The first time it renders the front face of the cuboid, and the second time it renders all its back faces. For each face, it stores the respective texture coordinates as colors. The result of such computation is shown in Figure 3.14, as illustrated in the "Acceleration techniques for GPU-based volume rendering" publication [13].

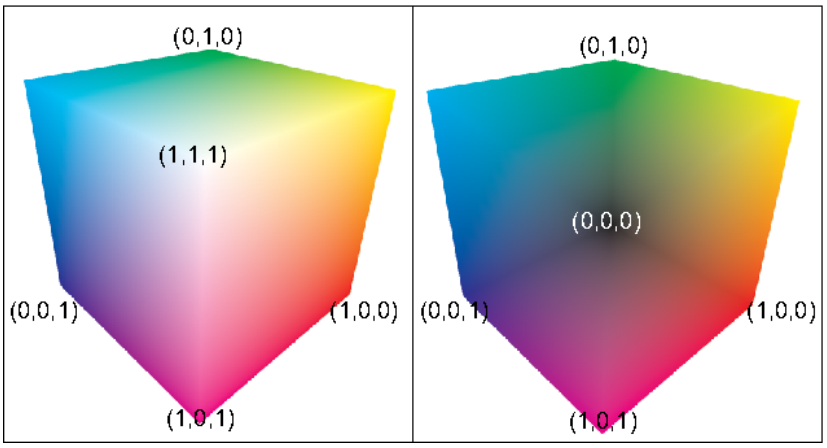

Figure 3.14: Ray marching volume definition

The end result is a good representation of the contents of the texture. But if the camera happens to get inside the rendering volume, the cuboid, as the initial position for the ray marcher is not defined, the visualization does not work as intended. Figure 3.15 illustrates the rendered volume from the outside and Figure 3.15 shows the problem that arises if the camera goes inside the said volume. The result is that the clouds look like they are crushed to the edges of the texture.

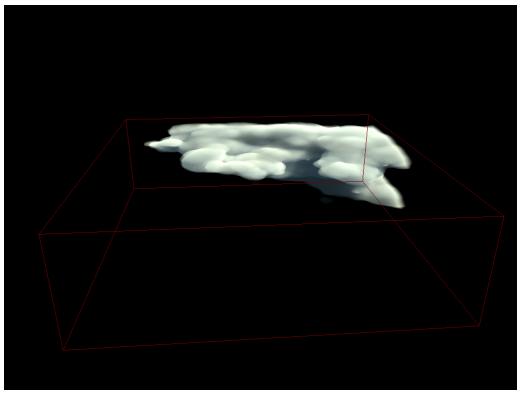

Figure 3.15: Outside visualization

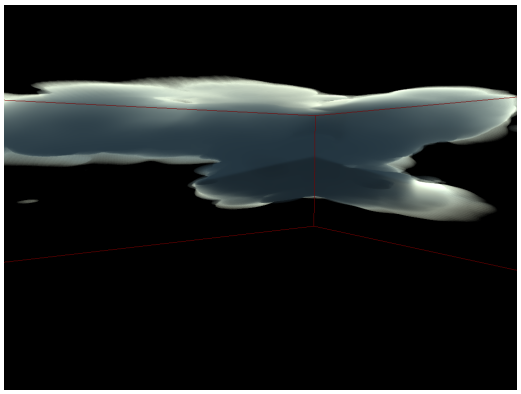

Figure 3.16: Inside visualization

Such an issue can be addressed in two ways. Allow the user of the engine to navigate inside the cloud by generating a billboard that is at all times in front of the camera to have always a starting point for the ray. Or by not allowing the camera to get into the rendered volume.

This technique is much faster than using billboards as described previously but is more time-consuming to implement and not as trivial as it may seem to implement and fully understand.

The engine is flexible and allows for the definition of a potentially infinite number of steps to sample the 3D texture. A low amount of steps can be defined to gain performance over quality. Meanwhile, a greater number of steps can be chosen to increase the quality of the rendered clouds.

In this case, the number of steps for the ray marcher has been set to 256. The sampling starts from a given start position into a computed distance that is equal to the normalization of the difference between the end and start position multiplied by the step size, and ends at a defined end position:

```
// Ray Marching variables:
layout (bindless_sampler) uniform sampler2D texture3;  // Cuboid front
layout (bindless_sampler) uniform sampler2D texture4;  // Cuboid back
int MAX_NR_STEPS = 256;
float STEP_SIZE = (1.0f / MAX_NR_STEPS);
...
void main() {
   ...
   // Direction vector:
   vec3 startPos = texture2D(texture3, texCoord.st).xyz;
   vec3 endPos = texture2D(texture4, texCoord.st).xyz;
   vec3 dir = endPos - startPos;
   vec3 deltaDir = normalize(dir) * STEP_SIZE;

  // Ray Marching loop:
  ...
}
```

Listing 3.2: Ray Marching

The ray marching iterative loop starts, and the final output fragment color is progressively computed. For each sampled point in the 3D texture, the coverage, height, and light values are used to determine their contribution to the final color of the currently computed fragment.

By doing that for each fragment on screen the final cloud modeling and lighting are achieved. The way light works and how it is computed will be discussed in the following section.

This class also features a noise threshold that can be modified at runtime to modify the visualization of the clouds, an effect of clouds expanding and reducing is in this way achieved. This is done by remapping the maximum and the minimum values and discarding the fragments with a noise value inferior to the current threshold value. The difference can be seen in Figures 3.17 and 3.18, this features the same clouds but with different threshold values. To try it yourself you can download the source code on GitHub [14].

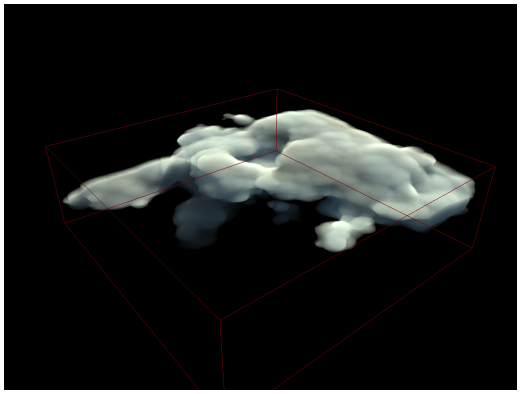

Figure 3.17: Low noise threshold value

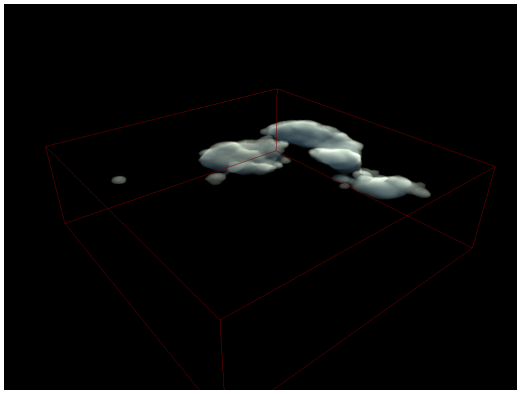

Figure 3.18: High noise threshold value

The movement of the clouds is computed in this class with the help of different threads to calculate in the background, while the main rendering loop is still active, the new coverage information. Changing at runtime the value of the coverage texture and recomputing the light allows the cloud to move. This is why the coverage and the light textures are by far the most important textures of this project.

To modify a texture at runtime, instead of calling the OpenGL code that creates a new texture, a special OpenGL function to modify existing textures is called, which can be seen in the code below:

```
1 const GLuint oglId = this->getOglHandle();
2    glBindTexture(GL_TEXTURE_2D, oglId);
3    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, getSizeX(), getSizeY(), GL_RGBA,
4                    GL_FLOAT, reserved->data.data());
5    glBindTexture(GL_TEXTURE_2D, NULL);
```

Listing 3.3: Modify texture at runtime

This, other than being logically correct, significantly helps with performance. As always creating and destroying textures at runtime is pretty slow as the GPU waits for the CPU to update the texture data.

3.3 Lighting

This section will focus on why lighting is so important for the rendering of clouds, will also show how lighting is implemented in the project, and will finally explain which kind of data structures are being used to do so.

3.3.1 The importance of light in cloudscapes

When dealing with cloudscapes, it is important to achieve a good level of realistic lighting, otherwise the clouds will just look like some gray blobs. The cloudscape rendered without lighting can be seen in Figure 3.19 and then render with lighting computed in Figure 3.20.

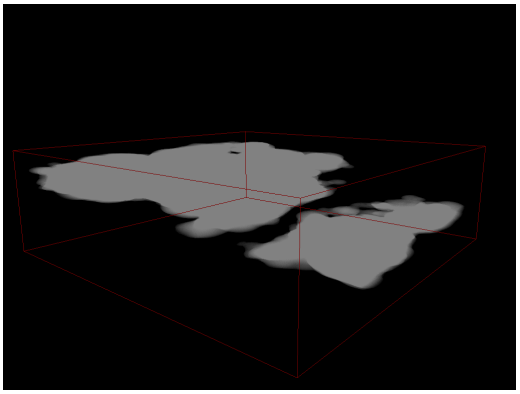

Figure 3.19: Unlit clouds

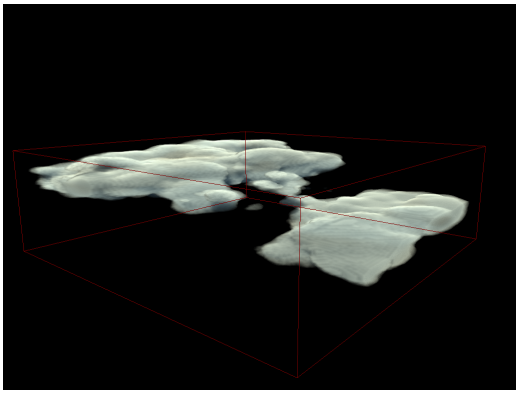

Figure 3.20: Lit clouds

Light helps give depth and thickness to the clouds, show their density with shadows, and bring out the little details that otherwise would pass unnoticed. The use of light is also necessary to make the clouds look good, clouds that are not affected by light would not be even remotely as appealing as lit clouds.

Seeing the shadow of a cloud on top of a lower cloud is impressive and helps in the overall immersion of the user in the scene. In this regard, the use of light also permits to change the color of the clouds as if they were affected, other than by the sunlight, also by the indirect lighting of the sky at different times of the day.

This type of effect can be seen taking life for a hypothetical sunrise, in Figure 3.21, and for a hypothetical cloudscape colored by the light contribution of the moonlight, in Figure 3.22.

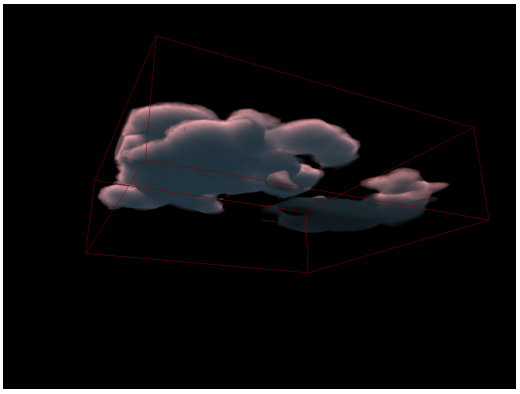

Figure 3.21: Sunrise

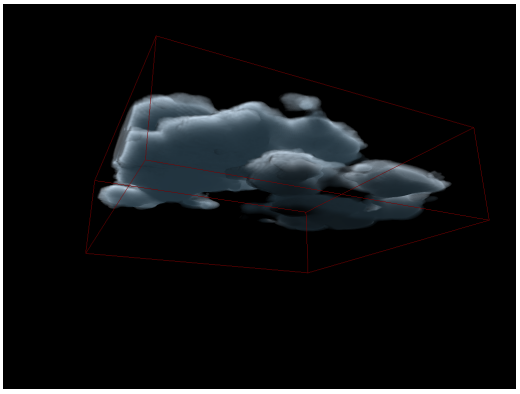

Figure 3.22: Moonlight

3.3.2 Light probes

To accurately compute lighting, it has been decided to make use of light probes. A light probe is an element in the scene that is responsible for collecting light information about light passing through a 3D space. In Figure 3.23 an example of what light probes would look

like, taken from a Stack Exchange post on the topic [15].

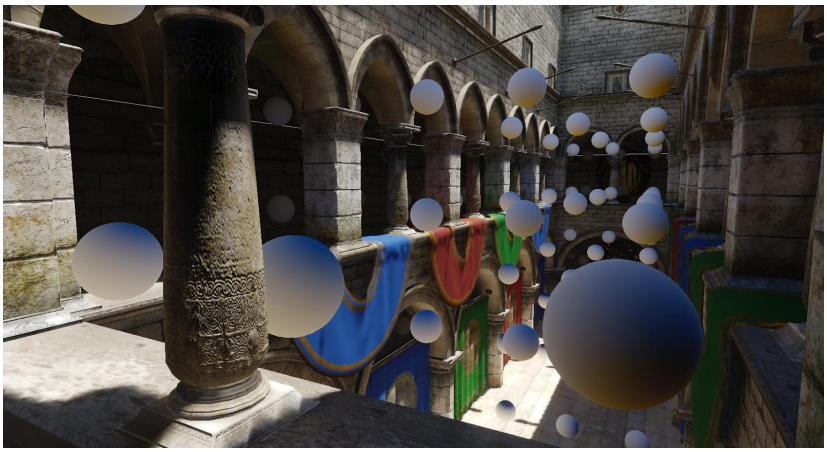

Figure 3.23: Light probes

For each voxel in the texture it has been decided to generate a light probe that would simulate the light absorbed by that particular voxel, and store such information in a three-dimensional texture. This computation is done by taking track of the amount of light that reaches a certain light probe that is not yet scattered into the cloud. To compute the ray that goes to the currently processed voxel ray marching is used once again.

A snippet of code that makes what has just been described can be seen right below:

```
for (int c = 0; c < MAX_NR_STEPS; c++)
   {
      // Getting current voxel color:
      float absorbedLight = getVoxelColor(voxelCoords) * absorption;
      if(absorbedLight >= noiseThreshold)
         absorbedLight *= remainingLight;
      else
         absorbedLight = 0;

      // Updating final color:
      remainingLight -= absorbedLight;
      if(remainingLight <= 0) {
         remainingLight = 0;
         break;
      }

      // Updating voxel:
      voxelCoords += deltaDir;

      // Updating distance:
      curLen += deltaDirLen;
      if (curLen >= dirLen)
         break;
   }
```

Listing 3.4: Generation of light texture

The code is written in such a way that the absorption factor of said cloud can be customized based on the specific needs. This allows us to play with this factor, and get the most out of every light situation, or time of the day.

Meanwhile, the getVoxelColor method is responsible for the sampling of the 3D noise texture to tell if in the current sampled point a cloud is currently present or not. If the current sampled point is not part of a cloud this method returns 0 effectively making the ray marching loop, shown just above, stop and therefore saving a significant amount of computational time.

3.3.3 Visualization of the lighting texture

In the development phase of the project, the visualization of the 3D texture was fundamental to make sure that lighting was correctly computed. To do that it has been decided to make use of the previously implemented billboards approach.

This time though, blending was not used because the goal was to be able to control the light behavior on the surface of the cloud. The “engine_pipeline_texture3dbillboards” class is responsible for the visualization of an arbitrary 3D texture, in this case, the lighting texture, making use of billboards. The result of such computations can be seen in Figure 3.24.

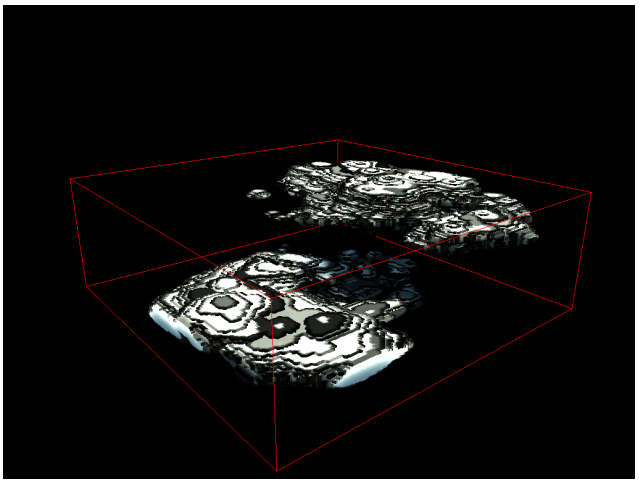

Figure 3.24: Lighting visualization

3.3.4 Rotating light source

In addition, to simply show the clouds lit once from a designated direction, it has also been decided that to give more liberty to the users of the engine, the rotation of the light source around the y-axis of the clouds is a good idea.

Figure 3.25 shows the clouds, lit from the default angle, that are generated as soon as the engine is initialized. Meanwhile, in Figure 3.26 the same clouds can be seen lit by a light source position that is slightly rotated towards the right side of the current camera position.

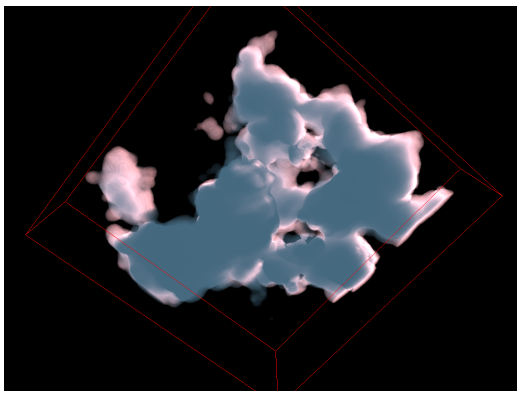

Figure 3.25: Default light source position

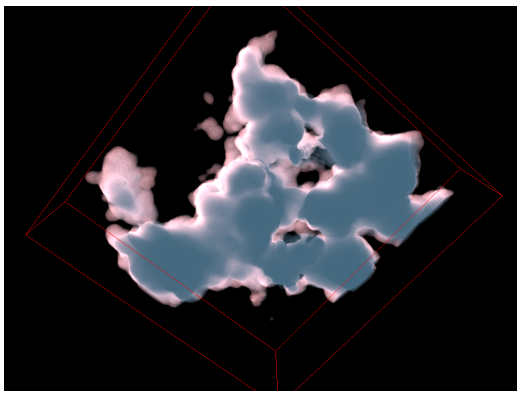

Figure 3.26: Rotated light source position

The angle of rotation can be defined by the user by using two designated keys that allows to increase or decrease its value. This approach also helps to make some parts of the clouds stand up more, it is clear that in the first picture the below cloud can be seen much easier thanks to the light source enlighting its edge, meanwhile, with the light source being located on its right side the two clouds seem almost to merge.

3.4 Skybox

This section will deal with the reasons that brought the need for a skybox in this project and the way this skybox is implemented.

3.4.1 Realistic sky

With the right use of different kinds of noises, the computation of lighting, and the update of the coverage at runtime some realistic and nice-looking cloudscape that evolves over time were achieved. But clouds are not supposed to lie in a black void, therefore the black background that was present until now needs to get replaced by some more accurate background. This is exactly where the use of a skybox comes into play.

The skyboxes that will be used in this engine are cloudless skyboxes, due to the nature of the engine as its main task is to produce three-dimensional clouds. In particular, the skyboxes are color gradients that aim to mimic the visual effect of a horizon.

This visual effect is also called atmospheric scattering and is generated by the light particles that are scattered in the atmosphere. This is visible when looking into the horizon as the particles of scattered light from the far portion of the atmosphere reach the human eye.

Skyboxes use a particular type of texture called cubemaps that, if wrapped into a cuboid, allows to display around the scene a real-life image or an illustration that depicts some kind of scenery. Figure 3.27 shows an example of a cubemap taken from a Medium article that speaks about skyboxes [16]. This type of cubemap that is also containing clouds, as previously said, will not be used in this engine.

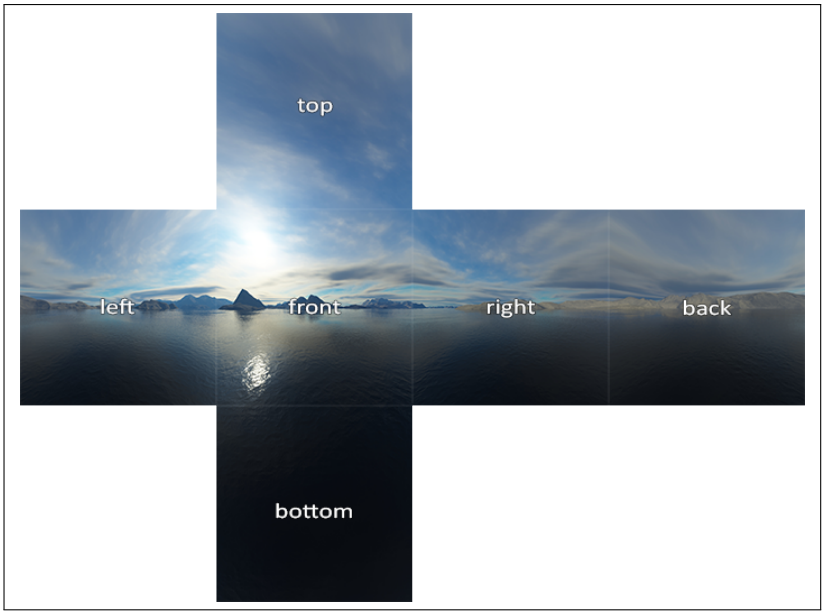

Figure 3.27: Skybox cubemap

3.4.2 Skybox implementation

The result of the skybox combined with the clouds, which can be seen in Figure 3.28, is unique and gives the demo a nice look in contrast to the black background featured before.

Figure 3.28: Skybox and clouds

The “engine_skybox” class is a simple class that enables the user of the demo to specify a set of images that will then be used to specify the different faces of the skybox cube and has a simple vertex and fragment shader that renders a cube on screen and apply the specified texture to it. It is important to point out that the skybox reacts to the rotation of the camera but is not reactive to the translation, therefore it is impossible to reach the skybox.

3.5 Graphical User Interface

This section will describe the technologies used to generate a Graphical User Interface or GUI, and the reasons why a GUI is fundamental to enhance the user experience.

3.5.1 FreeType

The implementation of OpenGL that this project uses, Graphics Library Framework, also known as GLFW, does not support text rendering as some other implementations as FreeGLUT does. This is due to the fact that GLFW gives much more control over context creation and window attributes, sacrificing the ease of use of implementations like FreeGLUT.

An external library for text-rendering is therefore required. Freetype [17] is the library of choice as it is able to load fonts, render them to bitmaps, and provide support for several font-related operations. What makes this library particularly useful is that it is able to load TrueType fonts.

A TrueType font, as thoroughly explained on LearnOpenGL [18] is a collection of character glyphs, one is shown in Figure 3.29 (image taken from [18]), not defined by pixels or any other non-scalable solution, but by mathematical equations. Therefore font images can be procedurally generated and no matter the size of the image, no quality will be lost.

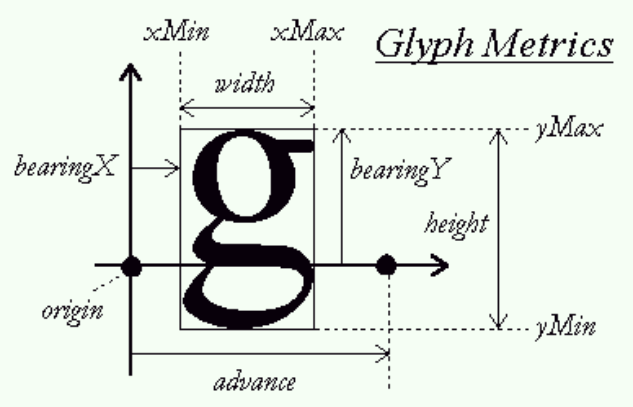

Figure 3.29: Character glyph

3.5.2 GUI Implementation

To implement correctly the GUI a dedicated class named “engine_gui” was created. This class is responsible for the loading of the first 128 ASCII characters when it is initialized for the first time, and for the rendering of the text. The text is rendered in a position and with a color, specified by the user.

The class uses in fact an orthographic projection in order to write text on the screen and makes use of shaders to color and correctly display on the screen the desired text.

Each character is saved into a texture that will, later on, be accessed, via its textureId which is stored into a map that has the character itself as key and the textureId as value. This makes it very easy for the rendering loop to retrieve the correct texture and display it in the correct position making use of VBOs where the information of where the character will be displayed is saved.

A close-up look at how the GUI on top of the rendered scene looks, can be seen in Figure 3.30. The use of such a GUI also allows the framerate of the engine to be seen in every machine, no matter if the currently used graphic card has a framerate counter built-in or not.

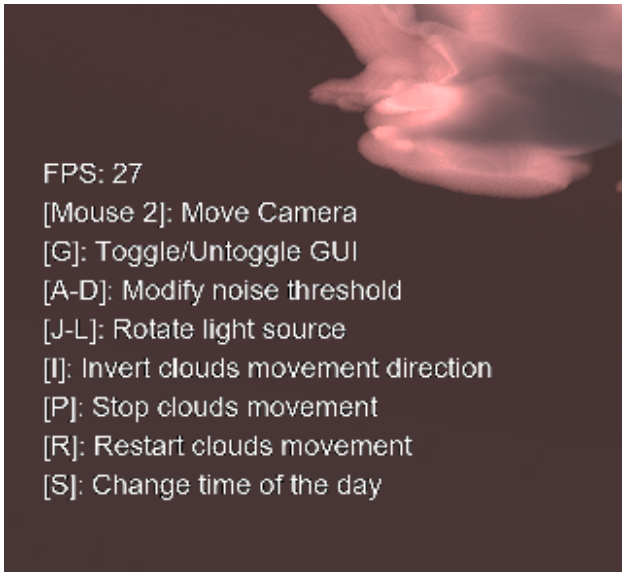

Figure 3.30: GUI close-up

3.6 Terrain

To show how this engine could potentially be used in the near future, a terrain was implemented. Such terrain was imported into the engine via the baseline code. However, its lighting and coloring have undergone some changes.

The terrain is affected by the same light that affects the clouds and its color changes according to the time of the day. We can see such terrain with the rendered clouds in Figure 3.31. The light also rotates according to the light that affects the clouds, giving the cloudscape and landscape continuity.

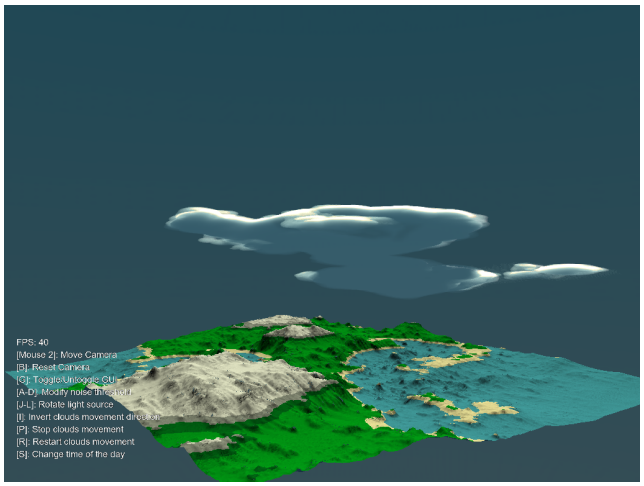

Figure 3.31: Terrain and clouds

The clouds do not currently cast shadows on the ground, but the implementation of such a feature would, as a consequence of the flexibility of the engine, only require an additional step in the rendering of the terrain. Ray marching would be computed for the terrain as well, and shadows would be cast if clouds were detected in the way.

Chapter 4

Results

In this chapter the final results achieved at the end of the project will be shown. In particular, the final modeling and lighting of the clouds, the use of a skybox and movement, and the performance of the product demo will be discussed.

4.1 Cloud looks

After a lot of testing with the generation of the noise 3D texture, needed to achieve convincing cloud shapes, the cloudscape got a much more chaotic look just as desired. The use of the 3 Worley noises with different textures allows the generation of some amazing details in the clouds.

All of this in combination with the properly computed lighting of the clouds allows seeing all those details well. The clouds themselves can be this way seen as composed of different layers. Figure 4.1 shows the lower frequency noise that defines the big chunks of the clouds as well as the higher frequency noise that defines the little bulges inside the clouds.

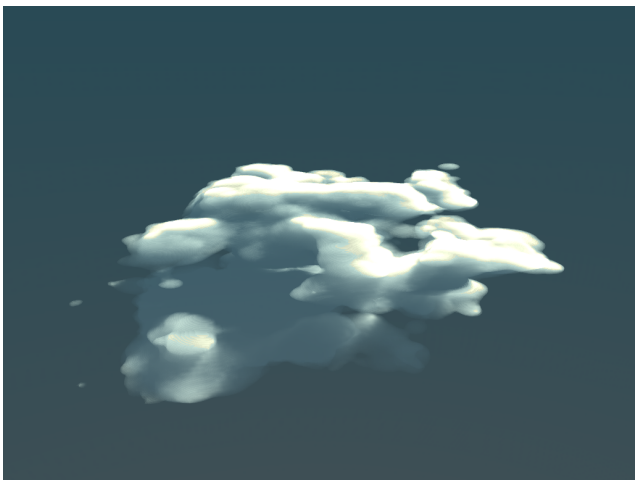

Figure 4.1: Cloud details

The use of a skybox to simulate a horizon, in combination with the movement of the clouds, brought some life to the cloudscape that was previously still and enclosed in a black void. This also gives the users of the application a realistic context in which clouds could eventually be inserted, giving overall a much more convincing look to the final product.

Finally, the clouds when observed from below, therefore the majority of the time, can be seen with their typical lit edges given by the lower density present there. This effect is shown well in Figure 4.2 where a much brighter color can be seen at the edges of the clouds compared to the area directly below it.

All these achieved effects are very similar to what was previously shown in the state-of-the-art chapter. The chaoticness of the clouds, their typical silver lining, and even the dark edges are all present.

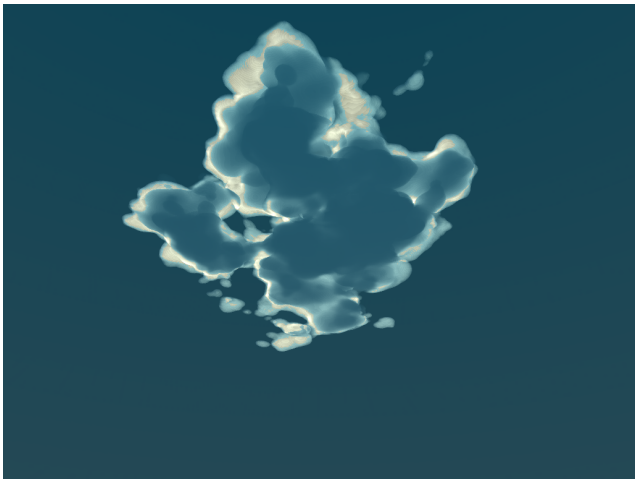

Figure 4.2: Cloud brighter edges

4.2 Movement and times of the day

As previously discussed in the implementation chapter, clouds do move. Their movement direction, or wind direction, can be specified at the start of the program. The use of a GUI would allow changing the direction value at runtime as the code that uses this information would not be affected by its change. However, the engine is so flexible that a weather system could be easily integrated in the near future allowing the coverage texture to be modified by the displacement of the weather in real-time.

The project does currently support the change of the current time of the day by pressing the designed keyboard keys. This allows going from a day visualization to a night visualization. The project currently features four different times of the day, sunrise, noon, sunset, and nighttime.

4.3 World coverage

The clouds cover a set amount of terrain in the virtual world, however thanks to the way the modeling textures were implemented, in particular thanks to the Perlin noise, the clouds are tileable and more terrain can be therefore covered if needed.

A potentially infinite amount of terrain can be covered with the use of these textures if they are carefully aranged side by side. Four textures could be this way placed in the world to cover a larger terrain and the engine is flexible and stable enough to let those texture tile with one another seamlessly, generating a bigger cloudscape with repeated patterns.

4.4 Performances

All the previously mentioned features can be rendered in real-time, thanks to the use of multiple threads. These threads do only modify the data in memory and do not upload the changes to the GPU as OpenGL is not capable of multi-threading. The changes are done by the main thread, which is also the one that holds the OpenGL context, after it successfully joined the other threads.

In fact with the use of an Nvidia Geforce GTX 970, a 2014 GPU, the scene is rendered at all times with an average framerate of above 24 FPS (Frames Per Second) which marks the start of real-time rendering versus the 1-5 FPS of interactive rendering.

Due to the large amount of data that has to be generated when the engine is initialized for the first time, it takes about 20 to 25 seconds, with an Intel(R) Core(TM) i7 4790k 4.00Ghz (8 Threads), to load the scene. This is because the filling of the noise 3D texture is done on the CPU. To let the user know of this a minimalistic loading screen was created as can be seen in Figure 4.3.

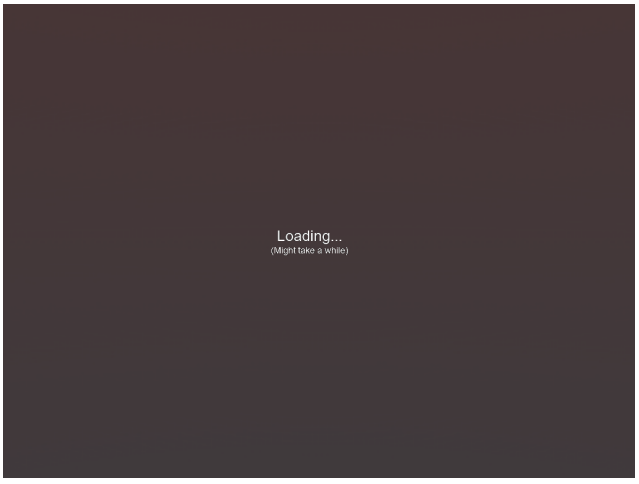

Figure 4.3: Loading screen

4.5 Testing

The baseline code, as mentioned in the state-of-the-art chapter, is a code that has been already tested thoroughly by the advisors. Therefore it was a safe code to start building on new features without the need for additional testing at the start of this project.

In computer graphics applications unit testing is often limited but proper unit testing was done on the compilation of the shaders and the loading of the data in the textures, which are the major elements in this project.

The various components of the built clouds model were thoroughly tested programmatically as there is no real parameter of what is right and wrong. The whole scene has to feel natural therefore half-automatic testing had to be done.

The engine was also tested with an AMD GPU, the Sapphire AMD Radeon r9 270x Vapor-X, to validate its use even with computers that do not make use of Nvidia GPUs. Further tests were also made on Nvidia laptop GPUs and were additionally done on an Nvidia Geforce GTX 1060, a GTX 1070, and an old Nvidia laptop GPU.

AMD GPUs tend to produce artifacts, which can be seen in Figure 4.4, those are generated from the sudden change of camera angle.

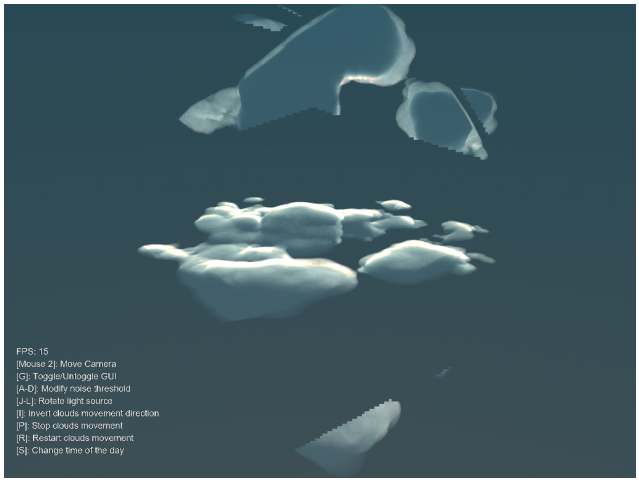

Figure 4.4: AMD - artifacts

Also due to the way AMD GPUs manipulate FBOs, the function glClearColor does not work as expected as it does not effectively clear the FBO's buffer content, producing once again weird visual artifacts that can be seen in Figure 4.5.

Figure 4.5: AMD - loading screen artifacts

4.6 User Interaction

Thanks to the implementation of a GUI, the users are almost immediately able to interact with the software as soon as the scene loads. The user will be able to modify all the major parameters of the scene such as the movement of the clouds, the rotation of the light source, and so on.

The GUI can also be toggled or untoggled just by pressing a button, giving the ability to acquire beautiful screenshots without the GUI in the way.

The best part of the engine is that flexible as it is the user can modify parameters and view the impact of said changes in the visualization of the clouds in real-time. The changes the user makes are, in fact, immediately reflected on what is shown on the screen making it actively participate in the visualization process.

Chapter 5

Conclusions

In this chapter the reached goals, the way the challenges of the project were faced during its development, and the future work that could be put into it to improve its current state will be discussed.

5.1 Conclusions

The generation of a cloudscape in real time using modern OpenGL is a really challenging topic to dive into as little to no documentation and examples are present online. This technology is relatively new, these techniques were used for the first time in 2015 for the realization of one of the greatest games of the last decade.

Great results were achieved in this project as all the cloud elements that Guerrilla Games featured are now fully functional in the engine. A lot of meetings with the advisors were needed to discuss the best approaches to deal with the different problems such a huge project could introduce.

Having however the opportunity to work with modern computer graphics and explore some of the newest techniques used to achieve the amazing visual results seen in modern videogames was really helpful and overall satisfying.

The required goals were successfully reached, a demo that demonstrates how such a cloudscape behaves in real-time was built, and even additional features were implemented. The state-of-the-art on the topic was broadly explored, and advanced techniques were implemented based on what was thoroughly explained in different papers on the topic.

5.2 Future Work

The project in its current state is complete and fully working without any particular problem. To furthermore enhance the realism of the cloudscape, which already looks really realistic, new noise can be implemented, just like the Perlin-Worley noise previously shown in the state-of-the-art chapter. This will allow the clouds to achieve even more realistic shapes.

Meanwhile, to enhance furthermore the user experience in the demo, which currently uses keyboard keys to change parameters, an interactive Graphical User Interface or GUI could be added. This would let the user change parameters such as light absorption, current coverage, and so on using the mouse.

To do so the use of an Immediate Mode GUI, or ImGUI, would be necessary. An example of such technology is the use that Ubisoft made of Dear ImGUI, an implementation of an immediate mode GUI, for Assassin Creed's Revelation and Rainbow Six Siege as explained in a Ubisoft Montreal article [19] that can be seen in Figure 5.1.

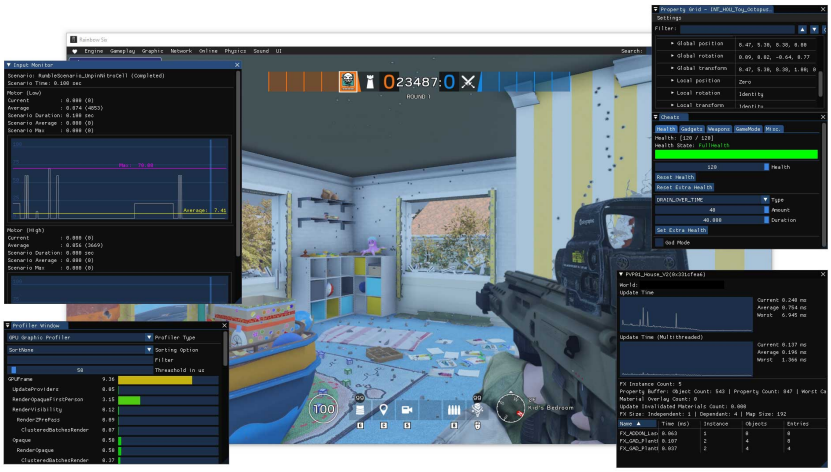

Figure 5.1: Rainbow Six Siege - Dear ImGUI

This cloudscape generation project could for sure benefit from a technology like Dear ImGUI for its future developments.

The last thing that could be implemented is a real and proper weather system that could simulate different weather such as rain, lightning, wind, and so on. That would give the cloudscape even more life, but that was not the goal of this project. As also pointed out in section 4.2 of the results chapter such a system could be implemented on top of the current engine thanks to its flexibility.

Bibliography

[1] A. Schneider and N. Vos. “The Real-time Volumetric Cloudscapes of Horizon Zero Dawn”. In: *SIGGRAPH, Advances in Real-Time Rendering* (2015) (cit. on pp. 2, 4–10).

[2] R. Wright Jr. G. Sellers and N. Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference 7th Edition*. Addison-Wesley Professional, 2015 (cit. on p. 2).

[6] P. Decaudin and F. Neyret. “Volumetric Billboards”. In: *HAL open science* (2011) (cit. on p. 11).

[13] R. Westermann J. Krüger. “Acceleration techniques for GPU-based volume rendering”. In: *Proceedings IEEE Visualization* (2003) (cit. on p. 22).

Sitography

[3] *E3 Website*. https://e3expo.com/. Accessed: 29-08-2022 (cit. on p. 3).

[4] *Horizon Zero Dawn - E3 2015 Trailer*. https://youtu.be/Fkg5UVTsKCE. Accessed: 29-08-2022 (cit. on p. 3).

[5] *SIGGRAPH*. https://www.siggraph.org/. Accessed: 29-08-2022 (cit. on p. 3).

[7] *LearnOpenGL - Deferred Shading*. http://www.lebarba.com/. Accessed: 01-08-2022 (cit. on p. 12).

[8] *Halo mod*. http://hce.halomaps.org/index.cfm?fid=2318. Accessed: 08-08-2022 (cit. on p. 13).

[9] *Lebarba*. http://www.lebarba.com/. Accessed: 09-08-2022 (cit. on p. 14).

[10] *LearnOpenGL - Blending*. https://learnopengl.com/Advanced-OpenGL/Blending. Accessed: 11-08-2022 (cit. on p. 19).

[11] *Ray marching & sphere tracing*. https://computergraphics.stackexchange.com/questions/161/what-is-ray-marching-is-sphere-tracing-the-same-thing. Accessed: 15-08-2022 (cit. on p. 20).

[12] *Michael Walczyk - Ray Marching*. https://michaelwalczyk.com/blog-ray-marching.html. Accessed: 12-08-2022 (cit. on p. 21).

[14] *GitHub - Source code*. https://github.com/AxelSalaris/realtime_volumetric_cloudscapes. Accessed: 01-09-2022 (cit. on p. 23).

[15] *Light probes*. https://computergraphics.stackexchange.com/questions/233/how-is-a-light-probe-different-than-an-environmental-cube-map. Accessed: 15-08-2022 (cit. on p. 26).

[16] *Aframe VR Skybox Configurator*. https://medium.com/hello-meets/aframe-vr-skybox-configurator-515c70df9ae1. Accessed: 15-08-2022 (cit. on p. 29).

[17] *FreeType*. https://freetype.org/. Accessed: 23-08-2022 (cit. on p. 30).

[18] *LearnOpenGL - Text Rendering*. https://learnopengl.com/In-Practice/Text-Rendering. Accessed: 23-08-2022 (cit. on p. 30).

[19] *Ubisoft Sponsors "Dear ImGui"*. https://montreal.ubisoft.com/en/ubisoft-sponsors-user-interface-library-for-c-dear-imgui/. Accessed: 15-08-2022 (cit. on p. 40).