

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1594

Volumetric Atmospheric Effects Rendering

Dean Babić

Zagreb, June 2018

Zagreb, 9 March 2018

MASTER THESIS ASSIGNMENT No. 1594

Student: **Dean Babić (0036470256)**
Study: Computing
Profile: Computer Science

Title: **Volumetric Atmospheric Effects Rendering**

Description:

Explore generation of atmospheric visual effects like clouds, fog or smoke. Investigate different rendering techniques such as texture blending and ray tracing for rendering volumetric atmospheric effects. Develop an application for rendering volumetric atmospheric effects at the wind farm Pometeno brdo virtual environment. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms.

Implement the appropriate software solution. Use the Unity development platform. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Issue date: 16 March 2018
Submission date: 29 June 2018

Mentor:




Full Professor Željka Mihajlović, PhD

Committee Chair:



Full Professor Siniša Srblić, PhD

Committee Secretary:



Assistant Professor Tomislav Hrkać, PhD

Zagreb, 9. ožujka 2018.

DIPLOMSKI ZADATAK br. 1594

Pristupnik: **Dean Babić (0036470256)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Volumetrijske tehnike u ostvarivanju prikaza atmosferskih učinaka**


Opis zadatka:

Proučiti postupke generiranja oblaka, magle, dima i ostalih vizualnih učinaka prisutnih u atmosferi. Proučiti tehnike postizanja ovih učinaka korištenjem stapanja prozirnih tekstura te korištenjem praćenja zrake. Razraditi proučene tehnike i ostvariti prikaz u kontekstu scene vjetroparka na Pometenom brdu. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama.

Izraditi odgovarajući programski proizvod. Koristiti programsko okruženje Unity. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:




Prof. dr. sc. Željka Mihajlović

Djelovođa:



Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:



Prof. dr. sc. Siniša Srblić

Acknowledgments

Firstly, I would like to thank my mentor, Prof. dr. sc. Željka Mihajlović for the time she dedicated towards my education and the effort she put into my projects.

I would especially want to thank my girlfriend for her support and help with proofreading my work.

Finally, I would like to thank my dear family and friends for their help, support and good times during all these years throughout my education.

Contents

1. Introduction	1
1.1 Cloud rendering.....	1
1.2 Practical solution	2
1.3 Restrictions and references.....	2
2. Cloud physics	4
2.1 Types of clouds	4
2.2 Behavior	5
2.3 Lighting.....	5
2.3.1 Absorption.....	6
2.3.2 Scattering.....	7
2.3.3 Extinction	8
2.3.4 Transmittance	8
2.3.5 Emission	8
2.3.6 Radiative transfer equation.....	9
3. Implementation	10
3.1 Modeling.....	10
3.1.1 Noise functions	10
3.1.2 Remap function.....	13
3.1.3 Cloud textures.....	14
3.1.4 Weather texture	17
3.1.5 Cloud shape definition	18
3.2 Lighting.....	20
3.2.1 Out-scattering	20
3.2.2 Directional scattering	21
3.2.3 In-scattering.....	22

3.3	Rendering.....	24
3.3.1	Ray marching.....	24
3.4	Tone mapping	26
4.	Optimizations	28
4.1	Cone sampling	28
4.2	Subsampling	28
4.3	Horizon culling.....	28
4.4	Cheap sampling	28
4.5	Early exit	29
4.6	Temporal reprojection	29
5.	Results.....	30
5.1	Hardware and Software.....	30
5.2	Resources	30
5.3	Performance.....	30
5.4	Visual results.....	33
6.	Conclusion	35
	Bibliography.....	36

1. Introduction

Realistic and impressive graphics is the goal of every game developer, to stay ahead of the competition and to offer the gamer the best gameplay. But they can push the graphics so far before computational power becomes an issue. With the advancements in hardware capabilities, we can use techniques that were previously only suited for offline rendering in real-time, but with some necessary optimizations and approximations.

One of those techniques is rendering of realistic and convincing cloud scenes. Realistic and convincing cloud scenes are not only the result of light scattering in participating media but also the result of dynamic clouds that can evolve over time, cast shadows and interact with its environment.

1.1 Cloud rendering

The area of cloud rendering is well researched in computer graphics. Many different techniques for rendering realistic clouds have been developed over the years, but they are often not fast enough to be used in real-time applications. One common solution is to use flat textures along with the skybox. This can produce very realistic clouds but only if the observer is placed at the ground level and is not expected to travel too far, such as first-person shooters and racing games. But in open-world games, a skybox would give a static feel and the sense of travel would be lost. By using 2D textures to render clouds we are losing the depth perception and the clouds appear flat. Also, it is possible to use a sequence of images to animate 2D billboards at least making clouds somewhat dynamic and interesting. But if we want better and far more immersive results volumetric clouds are needed.

The cloud system in this thesis is inspired by a recent technique developed by Andrew Schneider and Nathan Vos [4]. Their work showcased the ability to render clouds faithfully, with the ability to control their size, shape, speed, and lighting under 2 milliseconds. Furthermore, they improved the technique further by reducing the computational time, by introducing a better way to control the modeling, animating and lighting of clouds and by implementing weather simulation for the cloud system 'Nubis' that was originally made for the game 'Horizon Zero Dawn' [5].

1.2 Practical solution

The goal of this thesis was to investigate and implement a technique for rendering realistic clouds within Unity game engine in real-time. The rendering technique will be based on the cloud system 'Nubis' and will be able to produce clouds that are different in their shape, type, and density, while still being dynamic and evolve over time. The clouds should be rendered into a spherical atmosphere which would allow them to bend over the horizon. For the clouds to appear dynamic, we must render them in real-time. Real-time means that we are aiming for a frame time of about 33 milliseconds, effectively giving us a render frequency of 30 frames per second. The clouds are used to enhance the feeling of the scenery in the 'Pometeno Brdo' project which visualizes the Končar wind farm.

1.3 Restrictions and references

The approach was to analyze and assess different techniques to render volumetric clouds and to implement them in the Unity game engine with careful considerations of the render times. The solution was evaluated both by the time it takes to render and the number of resources required. In order to evaluate how realistic the clouds are, real photographs are used for comparison.

Photographs of clouds are presented to show the kind of effects we are trying to accomplish. Figure 1.1 shows how the sunlight scatters through the clouds, which results in bright edges and dark bottoms.



Figure 1.1 A photograph showing clouds with flat bottoms and rounded, wispy tops.



Figure 1.2 We can see how the light attenuates as it travels through the cloud

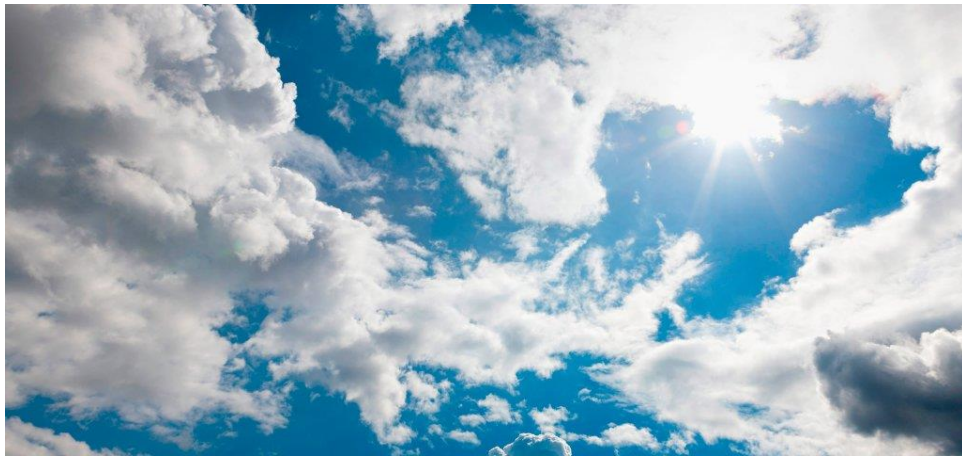


Figure 1.3 A photograph showing forward scattering in the direction of the sun



Figure 1.4 Packed round shapes like cauliflower, billows

2. Cloud physics

Water vapor is invisible to the human eye, and it is not until water condensates in the air, that clouds appear. Clouds are formed when the humid air rises and expands as it reaches lower atmospheric pressure. This is because when the air expands, the temperature falls, and the water vapor in the humid air condenses. The formation and structure of clouds are very dynamic and are mainly the result of vertical motions. The convection plays a big part in the features of the clouds.

Another aspect that contributes to the features of the clouds is the droplet size distribution. The droplet size distribution affects the Mie scattering, which accurately describes how the photons scatter inside the water droplets for different angles of approach [9].

2.1 Types of clouds

Clouds can appear in many different shapes and variations. Most cloud types are named after a combination of its attributes.

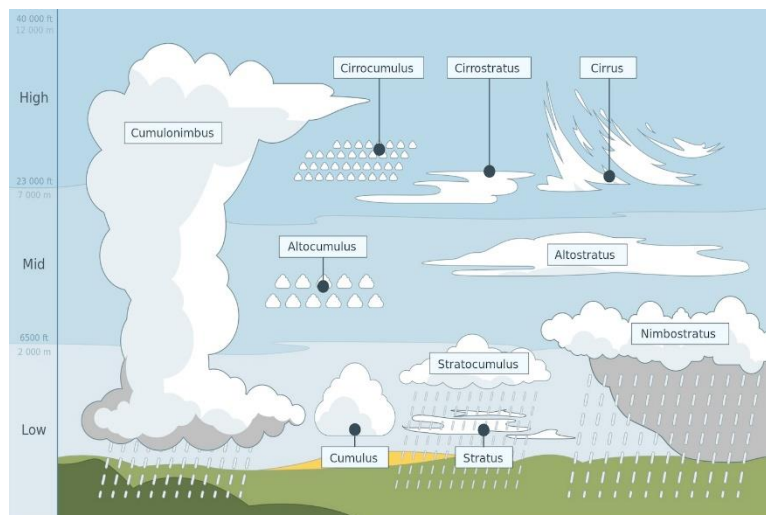


Figure 2.1 Cloud types and names

The attribute Cumulus is used for clouds that have a puffy appearance. Clouds that have the attribute Stratus appear as long flat layers. The attribute Cirrus is used to describe wispy stretched clouds that exist high in the atmosphere. The last common attribute is Nimbus which is used for clouds with precipitation.

Clouds below 1.5 km retained their names like Stratus and Cumulus, etc., clouds above 1.5 km but below 4 km in altitude earned the prefix 'Alto', giving us names like Altocumulus and Altostratus. Clouds above 4 km, in the top of the cloud zone, earned the prefix Cirro.

Warm water vapor creates Cumulus clouds and pushes them upwards into the atmosphere. This updraft is part of what is known as a convection current. When this current pushes the Cumulus clouds all the way to the Cirro layer we can often see a classic anvil shape. This happens because the warm water vapor hits the cold Cirro layer and is not able to continue rising but rather cools down, condenses and starts to fall.

2.2 Behavior

When modeling clouds it is important to follow their physical behavior for them to be realistic. Temperature and pressure are key components of how clouds form and behave. As water vapor rises with heat into the atmosphere, where it is colder, the water condensates and forms into clouds. Air temperature decreases over altitude and since saturation vapor pressure strongly decreases with temperature, dense clouds are generally found at lower altitudes. Rain clouds appear darker than others which is the result of larger droplet sizes. This is because larger droplets absorb more and scatter less light.

Wind is another force that drives clouds and is caused by differences in pressure at different parts of the atmosphere. Clouds can, therefore, have different wind directions at different altitudes. Since our focus is low altitude clouds close to the viewer, we assume that all these clouds move in the same wind direction.

2.3 Lighting

This section covers light behavior when traveling through participating media [2]. Participating media is a volume where refraction, density and/or albedo (ratio of reflected over incoming light) changes locally. Real world clouds do not have a surface that reflects the light, instead light travels through them. Photons interact with water droplets, that may absorb or scatter them, which causes a change in radiance.

Consider a single particle, molecule, dust, a water droplet, etc. as shown in Figure 2.2. Photons are going to interact with it and bounce off its 'surface'. In the case of clouds, imagine a single microscopic water droplet, the droplet will either refract or reflect the rays of light, but almost none of them will be absorbed; therefore, clouds are so bright, i.e. their albedo is nearly 100%.

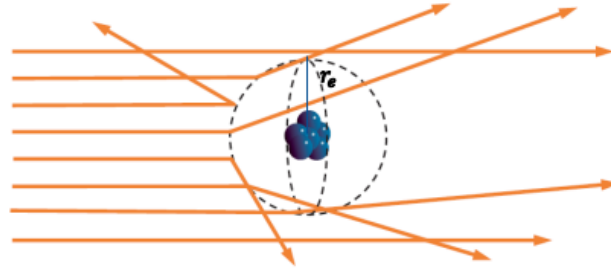


Figure 2.2 Light refracting off an atom

But when the medium is filled with a large density of such particles then it is possible that the light might get absorbed.

There are four different ways radiance may change in participating media, these four different ways are described in Figure 2.3. It can be due to absorption, in-scattering, out-scattering or emission.

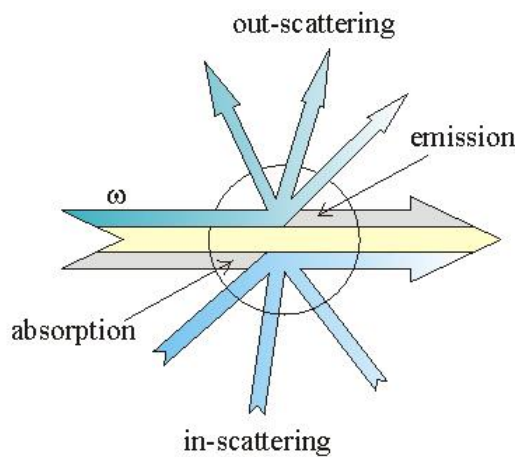


Figure 2.3 Four ways light can interact with participating media

2.3.1 Absorption

Absorption coefficient σ_a is the probability that a photon is absorbed when traveling through participating media. When a photon is absorbed, it causes a change in radiance. Reduced radiance due to absorption at position x when a light ray of radiance L travels along $\vec{\omega}$ is given by equation (2.1).

$$e^{-\sigma_a(x)dt} L(x, \vec{\omega}) \quad (2.1)$$

Rain clouds are generally darker because they absorb more light. This is because rain clouds have a higher presence of larger water droplets, which are more effective at absorbing light.

2.3.2 Scattering

Radiance may increase due to in-scattering or decrease due to out-scattering. The coefficient σ_s is the probability that a photon will scatter when traveling through participating media. Increased radiance due to in-scattering is shown in equation (2.2). In this equation $P(x, \vec{\omega})$ is a phase function, which determines the out-scatter direction from the light direction $\vec{\omega}$.

$$L_i(x, \vec{\omega}) = \int_{\Omega_{4\pi}} P(x, \vec{\omega}) L(x, \vec{\omega}) d\vec{\omega} \quad (2.2)$$

Many different phase functions exist and are suitable for different types of participating media. The phase function can scatter light uniformly in all directions as the isotropic or scatter light differently for every direction. For larger particles like pollutants, aerosols, dust and water droplets we must use Mie scattering. Mie scattering theory is very difficult to comprehend, the variety and complexity of shapes and behaviors of various components of the atmosphere usually makes phase function very difficult to work with.

For example, here is what an average statistical phase function of a cloud would look like:

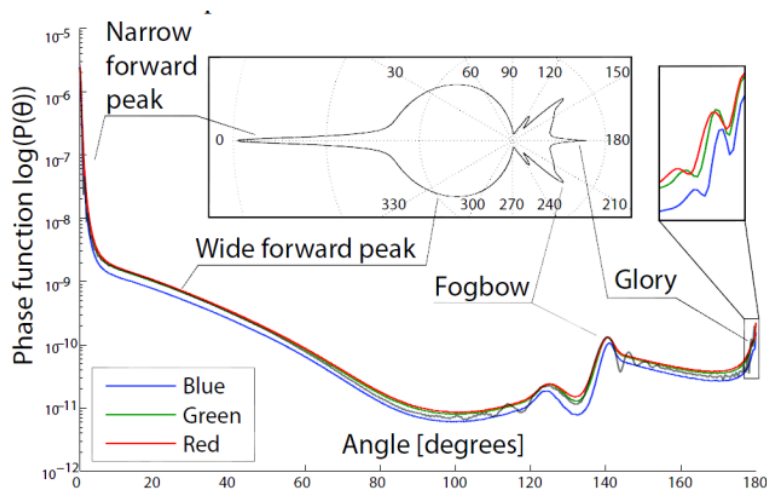


Figure 2.4 Mie scattering of a cloud

Clouds are generally white because they scatter light independently of the wavelength as opposed to atmospheric scattering which scatters blue wavelengths more than others.

2.3.3 Extinction

Extinction coefficient σ_t is the probability that photon traveling through a participating media interacts with it. The probability that a photon interacts and therefore causes a reduction in radiance is the sum of the probabilities for photon either being absorbed or out-scattered as described in Equation (2.3)

$$\sigma_t = \sigma_a + \sigma_s \quad (2.3)$$

2.3.4 Transmittance

Transmittance T_r is the number of photons that travel unobstructed between two points along a straight line. The transmittance can be calculated by using Beer–Lambert’s law as described in Equation (2.4).

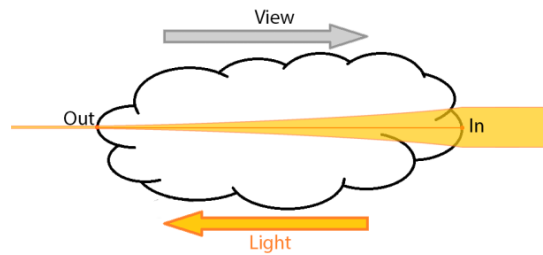
$$T_r(x_0, x_1) = e^{-\int_{x_0}^{x_1} \sigma_t(x) dx} \quad (2.4)$$


Figure 2.5 Intensity decrease through the cloud

So, any radiance Δ_x at distance entering the cloud will see its intensity decrease:

$$e^{-\sigma_t \Delta_x} L(x + \Delta_x \vec{\omega}, \vec{\omega}) \quad (2.5)$$

2.3.5 Emission

Emission is the process of increased radiance due to other forms of energy that have transformed into light. Increased radiance caused by an emission at a point x along a ray $\vec{\omega}$ is denoted by $L_e(x, \vec{\omega})$. Clouds do not emit light unless a light source is placed within, e.g. lightning, plane.

2.3.6 Radiative transfer equation

By combining the equations of the four ways that light can interact with participating media, it is possible to derive the radiative transfer equation through the law of conservation of energy. The radiative transfer equation is shown in Equation (2.6) and describes the radiance at position x along a ray with the direction $\vec{\omega}$ within a participating medium.

$$\begin{aligned} L(x, \vec{\omega}) = & T_r(x, x_s) L(x_s, -\vec{\omega}) + \int_0^s T_r(x, x_t) L_e(x_t, -\vec{\omega}) dt \\ & + \int_0^s T_r(x, x_t) \sigma_s x_t L_i(x_t, -\vec{\omega}) dt \end{aligned} \quad (2.6)$$

3. Implementation

This section shows how to create volumetric clouds within Unity game engine. It also covers resources required, how they are generated and used. First, we will discuss how to generate cloud textures by utilizing noise functions. After that, we need to see how to properly handle lighting of the cloudscares. Finally, we need to generate shader code to render the whole scene.

3.1 Modeling

The best way for us to approximate the shape and form of a cloud is through procedural generation. The typical way of rendering procedural clouds, or anything procedural for that matter, is to use noise functions, whether it be 2D clouds or volumetric 3D clouds.

3.1.1 Noise functions

This section covers the different noises that are used to create the cloud shapes and how these are generated. A combination of both Perlin and Worley noises are used to create clouds shapes. We pre-generate these noises in two different three-dimensional textures on the CPU and then use them in the shader.

3.1.1.1 Perlin noise

In 1985, Ken Perlin presented a technique for generating natural appearing noise, commonly known as Perlin noise [10]. Perlin noise is lattice-based and is generated by assigning a random gradient vector to each intersection of the lattice (Figure 3.1). When the value of the noise is read at a specific coordinate, the lattice cell wherein the coordinate lies is determined by rounding coordinates to the nearest cell. By utilizing modulo operation on all three axes we can determine where coordinate lies inside of the cell. Next, we need to calculate eight vectors from the given point to the surrounding eight points on the cell called distance vectors. Afterward, we take the dot product between two vectors, gradient and distance vector, which gives us influence values of each gradient vector. So now we need to interpolate between these eight values using the fade function because linear interpolation looks unnatural (3.1). Fade function for the improved Perlin noise implementation is:

$$6t^5 - 15t^4 + 10t^3 \quad (3.1)$$

Figure 3.2 shows what the resulting noise texture might look like. The texture may appear random, yet it has some structure making it suitable as a base texture of the clouds. By assigning the gradient vectors to the lattice in a repeating pattern, a wrapping texture can be attained.

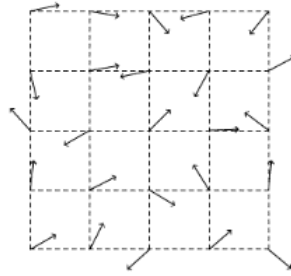


Figure 3.1 Perlin noise vector lattice

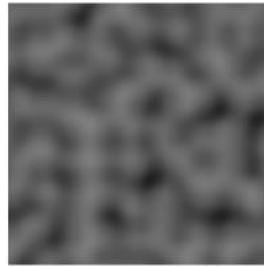


Figure 3.2 Perlin noise

3.1.1.2 Worley noise

Also known as Cellular noise or Voronoi noise, is a point-based noise as opposed to the lattice-based Perlin noise [11]. We use this noise type to create both wispy and billowing shaped clouds. The idea is to take random feature points in space and then for every point in the space assign the value which corresponds to the range to the closest feature point (Figure 3.3).

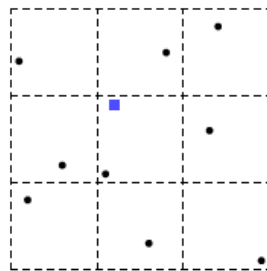


Figure 3.3 Worley noise feature points

But this approach is very slow especially so in three dimensions. Therefore, the naïve approach needs to be optimized as described in the steps below.

1. Subdivide a cuboid into equally sized cells.
2. For each cell, randomly place a feature point inside it. There must be only one feature point per cell.
3. For each point inside the cuboid, we apply color by measuring the Euclidian distance to the closest feature point and use that distance as a color.

The distance to the closest feature point is found by evaluating the feature point inside the current and surrounding 26 cells. By wrapping the edges, the texture will be tileable in all three dimensions.

This gives a result with ball-like features around each feature point which better imitates the look of a cloud. A cross-section of our 3D Worley noise is presented in Figure 3.4.

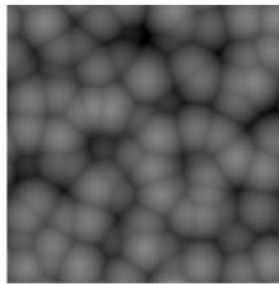


Figure 3.4 Worley noise

This algorithm can generate different octaves of Worley noise by changing the cell size. An octave is half or double of the current frequency and appears at intervals of 2^n . For example, generating Worley noise with four octaves and a cell size of two as starting frequency makes the following three octaves to be four, eight and sixteen.

3.1.1.3 Fractional Brownian motion

The noise textures described above form the foundation for our cloud textures. The next step of our noise generation is called fractional Brownian motion, often abbreviated as fBm, which when applied to noise textures renders results that look like smoke or clouds. The fBm noise is constructed by layering a noise texture of

different sampling frequencies (octaves) on top of each other. By changing the frequency in powers of 2, the wrapping effect can be preserved.

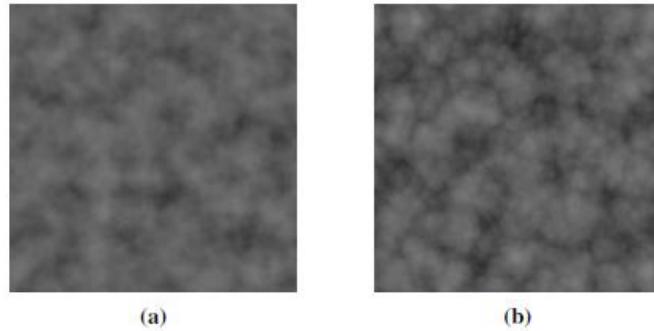


Figure 3.5 (a) Perlin noise with 5 octaves of fBm, (b) Cellular noise with 5 octaves of fBm

Compared to Figure 3.2 and Figure 3.4, we can see that the underlying structure is the same, but the overall noise texture has much finer details.

3.1.1.4 Perlin-Worley noise

Created by Andrew Schneider for use in 'Nubis' cloud system [4]. The noise is comprised of Perlin and inverted Worley noise layered on top of each other like the standard fBm approach. Worley noise is great for simulating tightly packed billow shapes as seen in Figure 1.4, but it lacks connectedness of the Perlin noise. Inverted Worley noise is used as an offset to dilate Perlin noise, this way they preserved qualities of both methods.

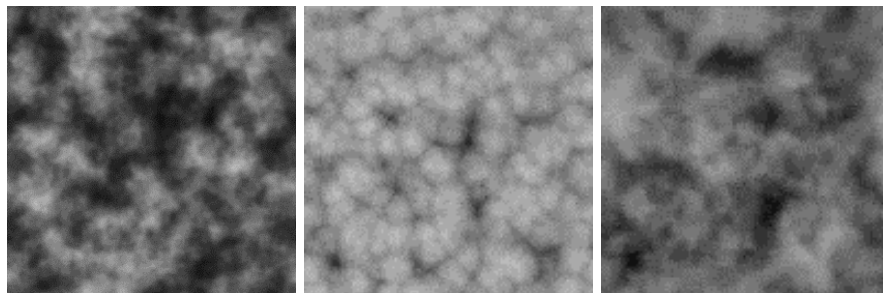


Figure 3.6 From left to right: Perlin, Inverted Worley, and Perlin-Worley

3.1.2 Remap function

Remap function is a function that is used ubiquitously in modeling and lighting of these clouds, reason being the fact it can prevent the loss of information as opposed to texture multiplying. The remap function simply takes a value, that lies inside one range $[\text{original_min}, \text{original_max}]$ and maps it to another range $[\text{new_min}, \text{new_max}]$, and is calculated like this:

Algorithm 1 Remap function

```
1: float remap(float value, float original_min, float original_max, float new_min,
2:   float new_max)
3: {
4:   return new_min + (((value - original_min) / (original_max - original_min))
5:     * (new_max - new_min));
6: }
```

3.1.3 Cloud textures

The noise generation algorithms presented above can be quite costly and the resulting textures are therefore precalculated and stored on the hard drive. To generate clouds as they are in ‘Nubis’ cloud system we need three textures. Two three-dimensional textures and one two-dimensional texture. It is always good to keep the resolution of the textures low in order to improve performance.

The first three-dimensional texture is used to create the base shape of the clouds. It has four channels, one with Perlin-Worley noise and three with different octaves of Worley noise. The resolution of this texture is 128^3 .

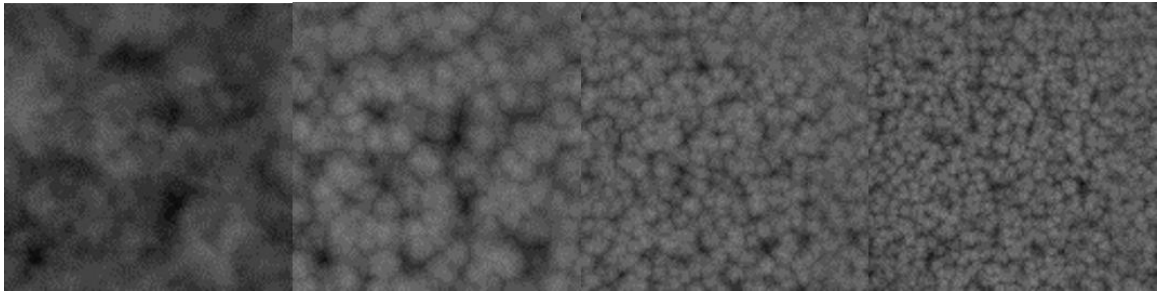


Figure 3.7 From left to right: Perlin-Worley, and three Worley noises at increasing frequencies

To combine these four channels, we use remap function because it prevents loss of too much density at the core of the base cloud shape.

If we use the below graph as an example, where the red line represents the base density of the cloud and the green line represents the high-frequency noise used to erode our cloud at the edges. If we performed a remap operation on the base density using the high-frequency noise as the new minimum value then we would not lose any density in the center of the cloud, which is exactly what we want.

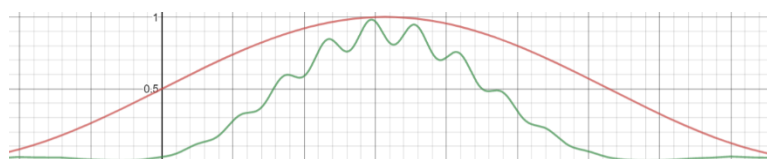


Figure 3.8 Base density in red, high-frequency noise in green

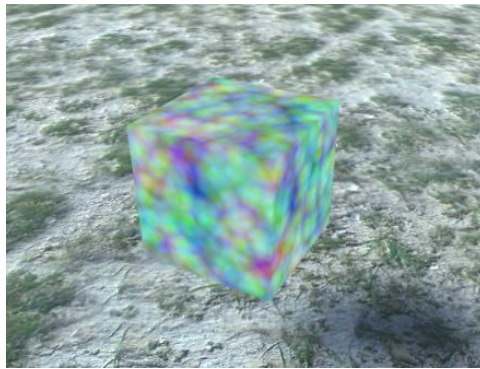


Figure 3.9 3D cloud base shape texture in-game



Figure 3.10 Clouds with just the base shape texture

The second three-dimensional texture is used for adding details to the base cloud shape and has three channels with different octaves of Worley noise. The resolution of this texture is 32^3 .

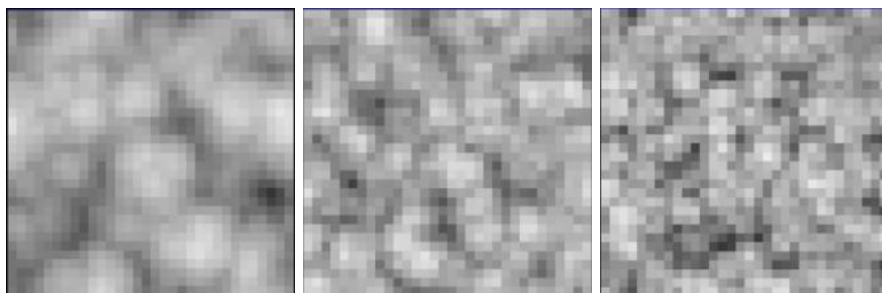


Figure 3.11 Three Worley noise textures at increasing frequencies

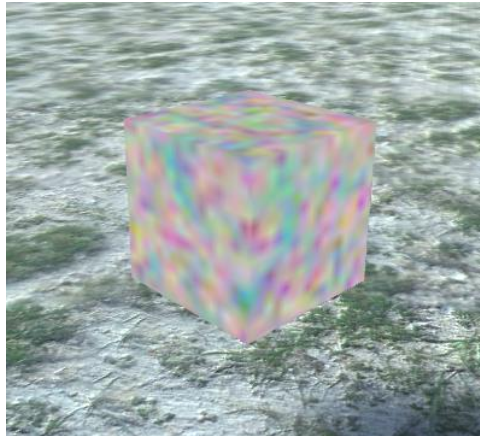


Figure 3.12 3D cloud detail texture in-game



Figure 3.13 Clouds after the detail texture has been added

To erode the edges of the clouds we need to use remap function again. We take a point inside of a cloud and simply check if the point lies at the edge of a cloud.

The third texture is used to add a sense of turbulence and has three channels with different frequencies of curl noise. Curl noise is nondivergent and is used to simulate fluid motion. The resolution of this texture is 128^2 .

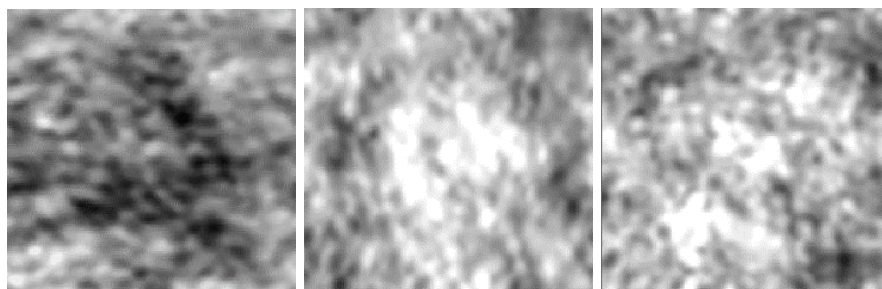


Figure 3.14 Three Curl noise textures at increasing frequencies

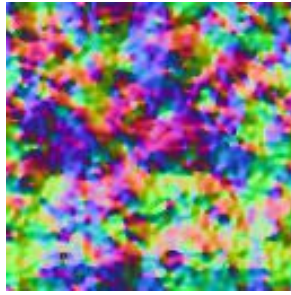


Figure 3.15 Resulting curl noise

3.1.4 Weather texture

Clouds are controlled by a repeating two-dimensional texture with three channels called the weather texture. This texture is repeated over the entire scene and is also scaled to avoid noticeable patterns of cloud presence and shapes. We can manipulate the scene with the simulation that progresses during gameplay.

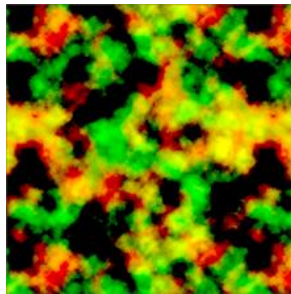


Figure 3.16 Weather texture

The red channel in this texture controls the coverage which is also used as the density of the clouds. The green channel controls the height of the clouds. A value of 0 will make the cloud have a height of 0 and therefore not be visible. If the height value is 1 this cloud will have maximum height value. The blue channel controls the type of cloud we want; where a value of 0 represents Stratus clouds, the value of 1 represents Cumulus clouds and value of 0.5 represents Stratocumulus clouds.

This texture controls the shape and type of the volumetric clouds in the scene. Volumetric clouds being ones between 1500 and 4000 meters, which means that Cirro clouds are not included. Cirro clouds are very distant and pretty flat on their own, so it is not necessary to render them volumetrically but instead, we can use textures.

3.1.5 Cloud shape definition

Instead of a single height gradient to change the noise signal over altitude, we can use three mathematical presets that represent the major cloud types (Figure 3.17). These values are stored in the weather texture and can be dynamically altered.

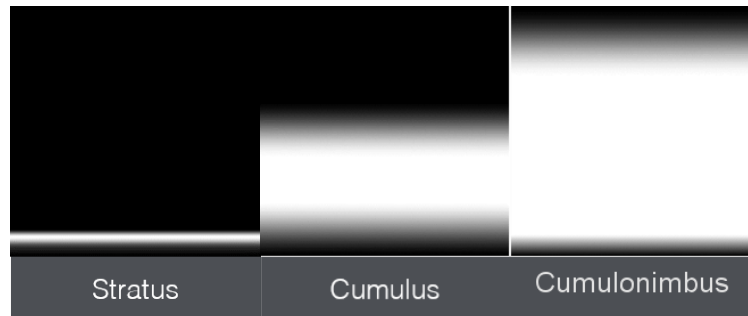


Figure 3.17 Height gradients for the different types of clouds

All clouds are modeled using the same approach and the density is independent of where in the atmosphere they appear. First, we create a basic cloud shape by sampling our first 3D texture. Then specific height gradient is applied. This height gradient lowers density at both the top and the bottom of the cloud as shown in Figure 3.17. Next, we multiply the result by the coverage value from the weather texture. The coverage value determines the density of the clouds. Thereafter, a height gradient is applied which lowers density at the bottom. The height gradient is implemented as a linear increase over altitude. The final density before any lighting calculation is done must be within the range (0, 1]. Densities that are zero or less after the erosion can be discarded since they will not contribute to any cloud shape. Densities larger than one are clamped to one to make lighting more balanced.

Algorithm 2 Cloud modeling

```
1: base_cloud *= density_height_gradient;  
2: float cloud_coverage = weather_data.r;  
3: float base_cloud_with_coverage = remap(base_cloud, cloud_coverage, 1, 0, 0);  
4: base_cloud_with_coverage *= cloud_coverage;
```

Defining a cloud shape only from one texture is not enough to get detailed clouds. Therefore, the smaller detail texture is used to erode clouds at the edges. The erosion is performed by subtracting noise from the 3D detail texture by using the remap function. The strength of erosion is implemented as a threshold value which only erodes densities lower than this value. We also distort this second noise texture by the 2D curl noise to simulate the swirly distortions from atmospheric turbulence.

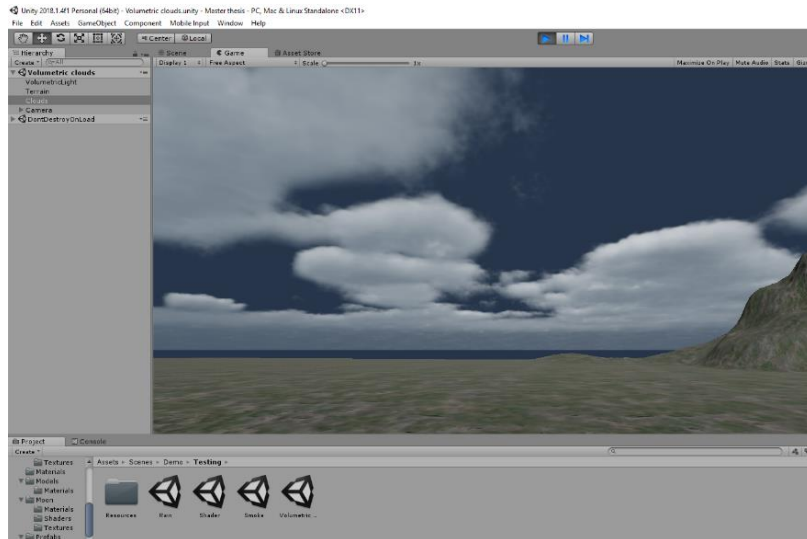


Figure 3.18 Rendering only Stratus clouds

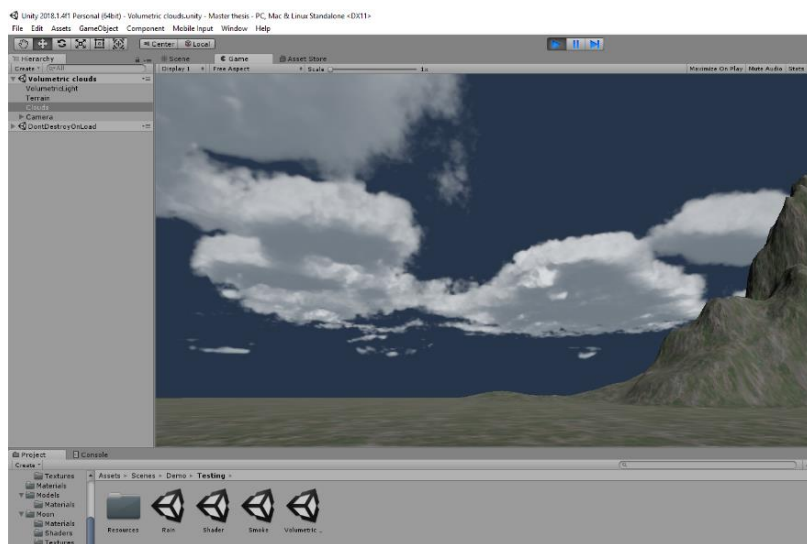


Figure 3.19 Rendering only Cumulus clouds

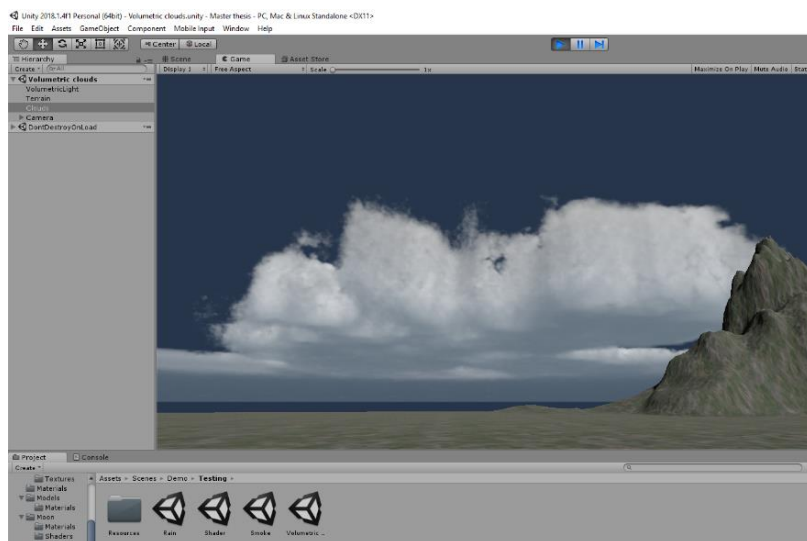


Figure 3.20 Rendering only Cumulonimbus clouds

3.2 Lighting

Cloud lighting is a very well researched area in computer graphics. The best results tend to come from a high number of samples. But for real-time applications that is not possible due to hardware limitations. Three approximation techniques are required to faithfully illuminate clouds.

3.2.1 Out-scattering

When light enters a cloud, the majority of the light rays spend their time refracting off water droplets and ice inside of the cloud before exiting cloud formation. By the time the light ray finally exits the cloud, it could have been out-scattered, absorbed by the cloud or combined with other light rays in what is called in-scattering as described in Section 2.3.

The attenuation of the light as it passes through the cloud can be faithfully approximated by using Beer-Lambert's law

$$T(d) = e^{-\sigma_t d}, \quad (3.2)$$

where T is transmitted light, σ_t is material dependent variable and d is the length the light travels through the material.

Beer's law states that we can determine the amount of light reaching a point based on the optical thickness of the medium that it travels through.

If we substitute energy for transmittance and depth in the cloud for thickness, we can see that energy exponentially decreases over depth. This forms the foundation of our lighting model.

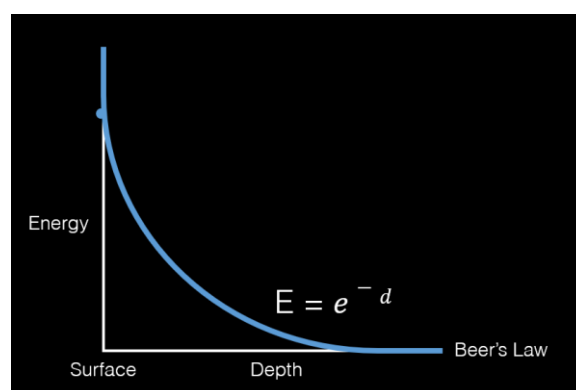


Figure 3.21 Beer's law curve



Figure 3.22 Only Beer-Lambert's law of illumination

3.2.2 Directional scattering

Directional scattering is the luminous property of the cloud. This is responsible for cloud lighting effects such as silver lining, fogbow, and glory. Due to the computational power limitations, we are not able to use Mie scattering phase function, which accurately describes the angular distribution of scattered light.

A common way of approximating the Mie scattering is to use the much simpler Henyey-Greenstein phase function. In 1941, the Henyey-Greenstein model was developed to help astronomers with light calculations at galactic scales, but today it is used to reliably reproduce Anisotropy in cloud lighting. The Henyey-Greenstein phase function is well suited for describing the angular distribution of scattered light in clouds as it offers a very high forward-scattering peak. The Henyey-Greenstein phase function is given as:

$$HG(\theta) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g \cos\theta)^{3/2}} \quad (3.3)$$

where HG is the magnitude of the scattered light, for a certain angle θ , and $g \in [-1, 1]$ is a parameter which determines the concentration of the scattering; a negative g gives backward scattering [8].

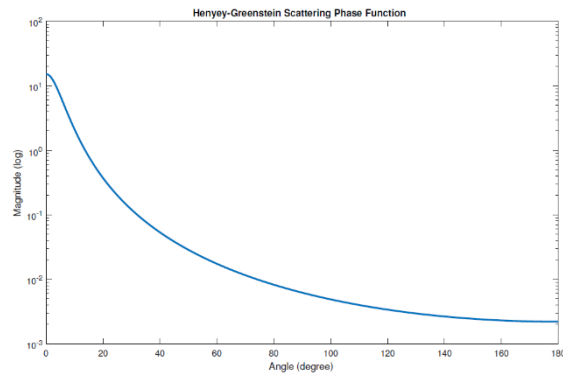


Figure 3.23 Henyey-Greenstein scattering phase function for $g = 0.9$

To calculate the cosine of an angle θ we need a dot product of a normalized light vector and normalized view vector.

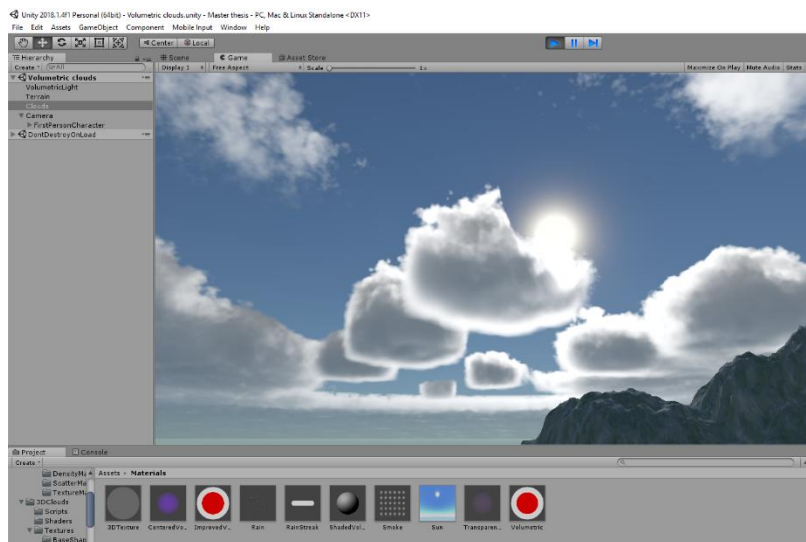


Figure 3.24 Beer-Lambert's law with Henyey-Greenstein phase function

3.2.3 In-scattering

This produces the dark edges and bases on the clouds. The dark edges on the clouds as shown in Figure 1.4 is something that is not as well documented with solutions, so Andrew Schneider and his team had to tackle the problem.

In-scattering is when a light ray that has scattered in a cloud is combined with others on its way to the eye, essentially brightening the region of the cloud you are looking at. In order for this to occur, an area must have a lot of rays scattering into it, which only occurs where there is cloud material. This means that the deeper in the cloud we are, the more scattering contributors there are, and the cloud is brighter. Consequently, the amount of in-scattering on the edges of the clouds is lower, which

makes them appear dark. Also, since there are no strong scattering sources below clouds, the bottoms of them will have fewer occurrences of in-scattering as well.

The reason we do not see this effect automatically is that our transmittance function is an approximation and does not take it into account. Therefore, we must approximate the in-scattering by using the following function:

$$S(d) = 1 - e^{-cd} \quad (3.4)$$

where S is the intensity of the scattered light, c is a variable that determines how fast the scattering effect builds up in the cloud, and d is the length the light travels through the cloud. This function together with Beer-Lambert's law constitutes our light function called Beer's-Powder approximation method.

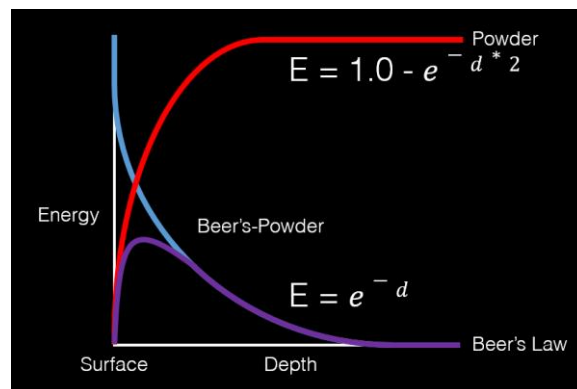


Figure 3.25 Beer's-Powder approximation method

We can see how a light ray, that passes through a cloud, first starts to scatter but as the cloud attenuates the light, the brightness falls off.

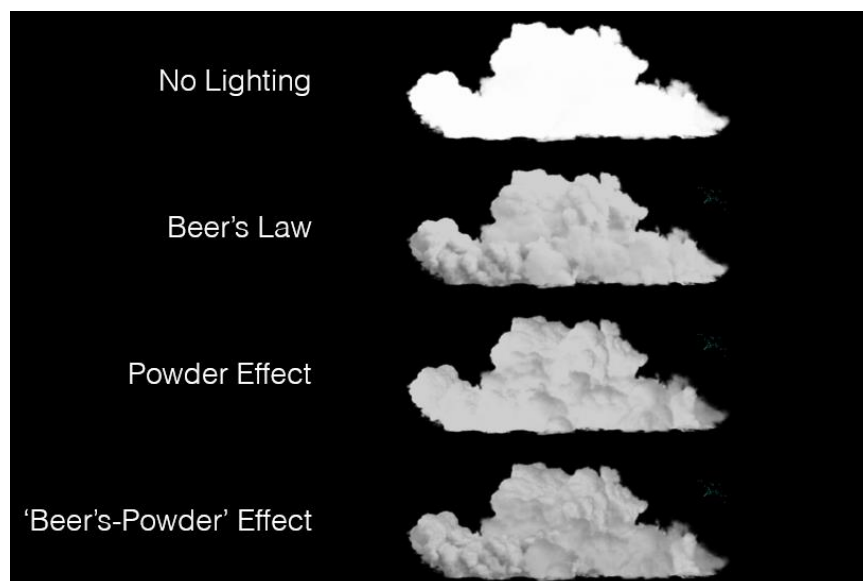


Figure 3.26 Difference in cloud appearance when utilizing different methods

Figure 3.26 shows how different lighting methods affect the cloud appearance. If uniform lighting is applied, we can discern from the shape that what we see is a cloud, but the cloud lacks all its features. When only Beer's law is used, we can see cloud features but the dark edges on the top of the cloud are missing. Powder effect enhances areas where there is very little cloud material, but incorrectly highlights the bottom of the cloud. Beer's-Powder effect both highlights the dark edges and correctly simulates transmittance in the cloud.

3.3 Rendering

To render clouds realistically, we would need to trace each photon radiating from the sun, follow their paths as they scatter through the clouds and register those few photons that end up in the camera. This is of course computationally very intensive, and we do not want to trace all photons that never end up in the viewer's perspective. Therefore, our cloud rendering algorithm is using a ray marching technique, which means that we reversely trace a ray from the camera into the scene instead.

3.3.1 Ray marching

Ray marching is the volumetric rendering technique that we use to render volumetric objects in our scene, volumetric objects in our case being clouds. When we use this technique, the scene is rendered by iterating through every pixel on the screen, and for each pixel, we march along a ray in the direction of the view vector, at each sample point along that ray we evaluate density and lighting. More sample points yield better results but are in turn more expensive.

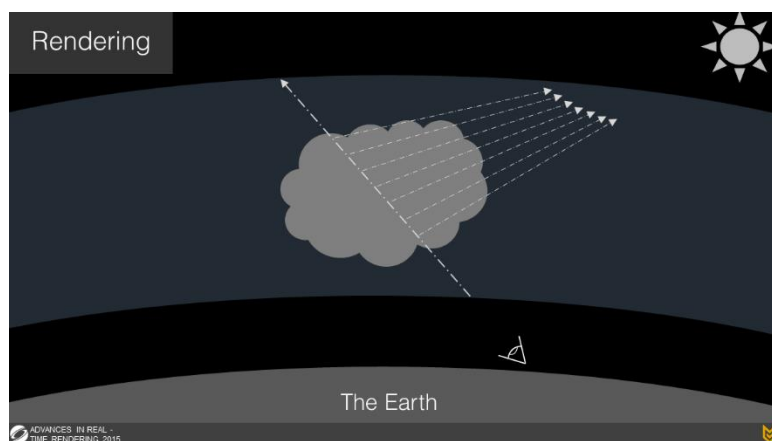


Figure 3.27 Ray marching

The rays that are cast from the camera advance by a constant step and evaluate density samples along the ray. The clouds are procedurally generated and stored in 3D textures that are placed in the scene. When we evaluate a point along the ray and determine it has a non-zero density value we know we are inside a cloud. Now, to give this point in the cloud some color we can light it by shooting a ray towards our single light source, the sun, and use the resulting energy information to color that point. This means that we are only considering the first order of scattering.

In our ray marching algorithm, we can set a maximum number of steps before algorithm stops and moves to the next pixel, this way we can stop the algorithm from continuing indefinitely in one direction. We can limit the number of steps we need to take along the ray further by setting the bounds in our algorithm. As shown in Figure 3.28 we can set lower atmosphere layer at a distance of 1500 meters, and upper atmosphere layer at a distance of 4000 meters. This way we only need to march between those two points along the ray.

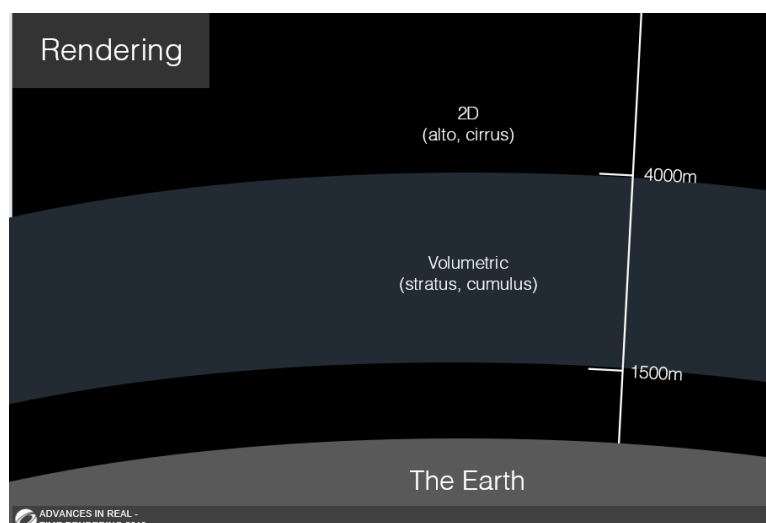


Figure 3.28 Atmosphere layers

Instead of sampling cloud density and illumination along the second ray towards the sun for each sample point of our first ray, we can use Cone sampling (Figure 3.29). Cone sampling is a more efficient way of determining the light energy that will be received by that point. Cone sampling involves taking some number of samples from inside the volume of a cone that is aligned with our light source. We take six samples using Cone sampling and make sure to have at least one placed relatively far. This far-away sample is a way of taking into account if the cloud and hence point we are trying to light are occluded by another cloud in the distance. Using these six

samples from within the cone we get a density value which is used to attenuate the light energy reaching the point we are trying to color.

Algorithm 3 Cone sampling

```
1: float SampleCloudDensityAlongCone(p, ray_direction)
2: {
3:     float density_along_cone = 0.0;
4:     for(int i=0; i<=6; i++)
5:     {
6:         // add the current step offset to the sample position
7:         p += light_step + (cone_spread_multiplier * noise_kernel[i] *
8:             float(i));
9:         // sample cloud density the expensive way
10:        int mip_offset = int(i * 0.5);
11:        density_along_cone += SampleCloudDensity(p, weather_data,
12:            mip_level + mip_offset, false);
13:    }
14:    return density_along_cone;
15: }
```

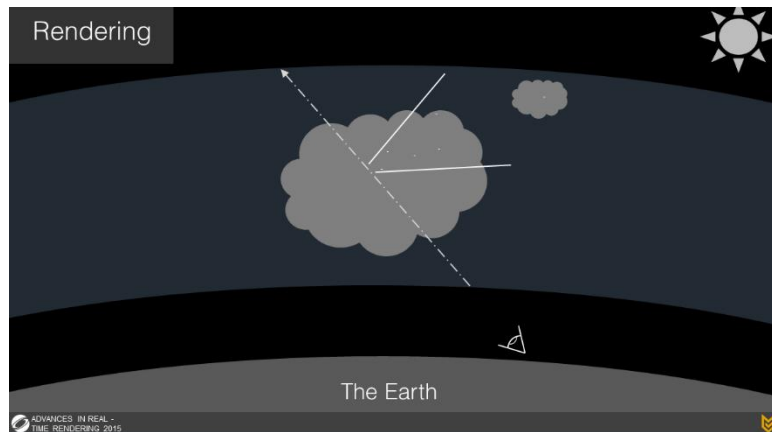


Figure 3.29 Cone sampling along the ray

3.4 Tone mapping

Tone mapping is a technique used to map one color space into another to approximate the appearance of high dynamic range images because displays and monitors have more limited dynamic ranges. Usually, the color channels in the framebuffers are stored as 8-bit values, ranging from 0 to 1. This means that the color channels can only assume 256 different values. This is all right for most applications, but the nature of the clouds and sun gives us very bright areas on the screen. This can lead to parts of the screen losing important color information as the colors get rounded to the nearest of the 256 values. A way around this problem is to use high dynamic range (HDR) lighting. Instead of binding textures with 8-bit values for each color channel, a floating-point value is used. HDR lighting also

allows us to assign color values above 1 to the framebuffer. Before the final framebuffer is rendered to the screen, the values need to be transformed back to the range [0, 1]. The Uncharted 2 tone mapping technique was implemented because it is very good and has become quite popular.

Algorithm 4 Uncharted 2 tone mapping

```

1: float3 ToneMapping(float3 x)
2: {
3:     const float A = 0.15;
4:     const float B = 0.50;
5:     const float C = 0.10;
6:     const float D = 0.20;
7:     const float E = 0.02;
8:     const float F = 0.30;
9:     return ((x*(A*x + C * B) + D * E) / (x*(A*x + B) + D * F)) - E / F;
10: }

```

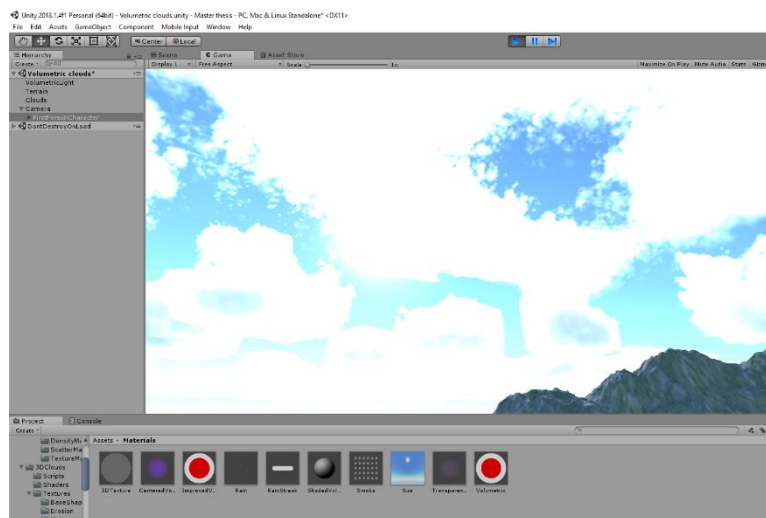


Figure 3.30 Rendered scene without tone mapping

In Figure 3.30 we can see how a non-HDR compliant monitor reacts to an HDR image; colors are rendered incorrectly, and we lose important cloud details.

4. Optimizations

In order to achieve real-time performance for this cloud rendering technique, several optimizations techniques have been implemented. Without these optimization techniques, our render times would be greater than 100 milliseconds, basically rendering our approach to the volumetric rendering of clouds useless.

4.1 Cone sampling

As described in Section 3.3.1, instead of using second ray march to determine the illumination of the point inside of the cloud we can use Cone sampling. Cone sampling is a more efficient way of determining the light energy that will be received by that point because we randomly choose several points in the general direction of the sun and approximate the lighting based on that.

4.2 Subsampling

Subsampling is a technique of decreasing the render targets resolution by the power of two. It can greatly reduce render time of a scene, and thus easily provide real-time performance. By further lowering resolution we could gain even greater performance boost but at the cost of a blurrier result.

4.3 Horizon culling

Because we cannot see clouds below the horizon line, we can exclude these clouds from rendering. Also due to the tiling nature of our base shape texture, when the cloud sphere drops below the horizon it starts to tile noticeably.

4.4 Cheap sampling

If we are not inside of the cloud we do not need to apply high detail noise, sample illumination, or for that matter apply any effect until we find ourselves inside of a cloud. Also, when we are doing Cheap sampling we also increase our step size while ray marching. Sampler does cheap work unless it samples density that is different than zero, in which case it starts using all the rendering techniques mentioned before while also decreasing step size. If the sampler does not sample

any density greater than zero for ten consecutive steps it again performs Cheap sampling.

4.5 Early exit

When we sample points along the ray we accumulate density value, i.e. increment alpha value. When the alpha value reaches 1 we do not need to keep sampling, so we stop the ray marching early.

4.6 Temporal reprojection

Temporal reprojection is a technique that uses pixels from the previous frame to color the pixels in the current frame. This is done by projecting the old frame onto the current frame. We calculate how much the camera has moved between two frames.

This technique is implemented as follows:

1. We store our previous frame in a texture.
2. We store the old camera information.
3. For a given pixel in the current frame, ray cast using the ray created by the current camera and the current pixels uv coordinates.
4. The ray cast will intersect with the sphere representing the inner layer of the atmosphere. The point of intersection gives us a world space position.
5. This world space position is then converted all the way back into uv coordinates using the old camera's view and projection matrices.
6. This uv coordinate if in the range from $[0,1]$ can be used to sample the pixel in the old frame that will be used to fill in the pixel in the current frame.

Using this reprojection technique we can get away with only ray marching for $1/16^{\text{th}}$ of the pixels in the current frame. The other 15 of the 16 pixels are filled in using reprojection.

This technique is the main reason the volumetric clouds were able to become a reality for the real-time applications. It improved the shader speed by as much as ten times making it feasible for use in game development.

5. Results

In this section, we will focus on the performance and visual aspect of this rendering technique which requires few precalculated resources. The performance aspect is measured in an empty scene containing only the terrain collider. Additional objects and scripts in the 'Pometeno Brdo' scene makes rendering even more challenging and pushes our rendering times higher. The visual aspect is focused on the clouds placed in the 'Pometeno Brdo' scene.

5.1 Hardware and Software

All results are captured on a Windows 10 machine with an Intel Core 2 Quad Q9550 CPU running at 2.88 GHz, 8 GB of DDR2 RAM and a Nvidia GTX 1050 Ti 4GB card running at 1800 MHz. The resolution of the application was set at 1920 x 1080.

All the measurements were captured in the Unity profiler window which includes the ability to monitor CPU, GPU, and RAM usages together with the ability to specifically say which piece of the code is the bottleneck.

5.2 Resources

The implementation of this cloud rendering technique requires few precalculated resources that are needed to achieve real-time performance.

Texture	Dimensions	Format	Size
Cloud shape	128 x 128 x 128	RGBA32	8192 KB
Cloud detail	32 x 32 x 32	RGB24	96 KB
Curl noise	128 x 128	RGB24	48 KB
Weather	512 x 512	RGBA32	1024 KB

Figure 5.1 Required resources

The total size of all these files is 9360 KB, i.e. 9.1 MB.

5.3 Performance

Most computations are performed in the pixel shader on the GPU, therefore the whole scene is GPU intensive. Although CPU is not utilized as much, it can still

present a problem. Older CPUs, like the Core 2 Quad series, which were released back in 2008, cannot compare with modern day CPUs in regards with instructions per clock (IPC) and memory latency, all of which can affect the performance of a GPU in a way.

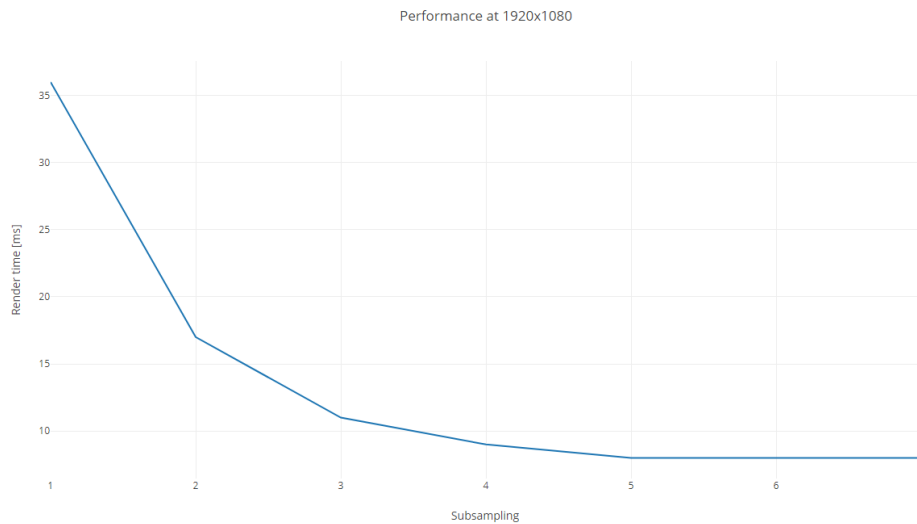


Figure 5.2 Run-time performance as a function of subsampling

Subsampling is a technique where we decrease render target resolution, and with the decreased resolution, it is natural to expect that the render time decreases.

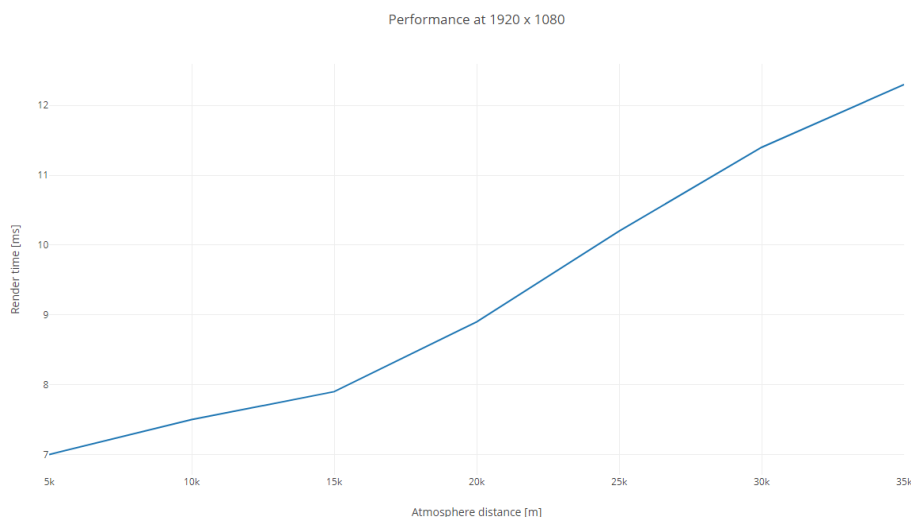


Figure 5.3 Run-time performance as a function of atmospheric distance

Short atmospheric distance means that the horizon is closer to the observer, which in turn means that our rendering algorithm more often finishes before it reaches the second layer of the atmosphere, therefore, performance is better at closer distances.

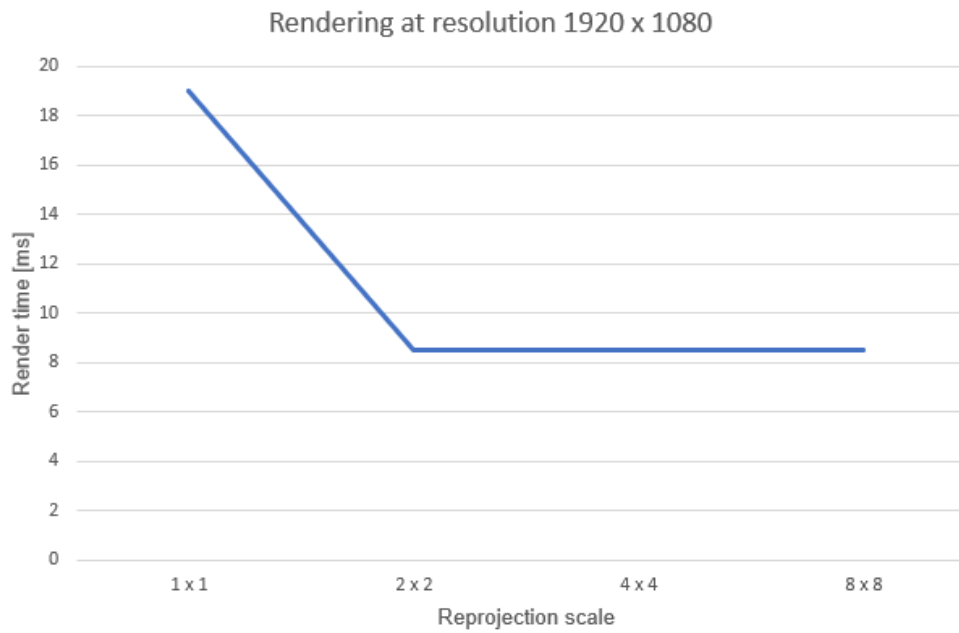


Figure 5.4 Run-time performance as a function of reprojection scale

At the base settings with lots of optimizations in place, we can see that reprojection can only benefit render time so far after which it loses on the effectiveness and starts to add perceived motion blur.



Figure 5.5 Run-time performance as a function of cloud coverage

When coverage is high, almost the entire sky is covered by the clouds and when coverage is set to low only a small part of the screen has clouds in it. As expected a low coverage is much faster mostly due to the early exit.

5.4 Visual results

Photographs are taken during gameplay. The scene is comprised of volumetric clouds, dynamic lighting, and wind generators.



Figure 5.6 Yellow sunset

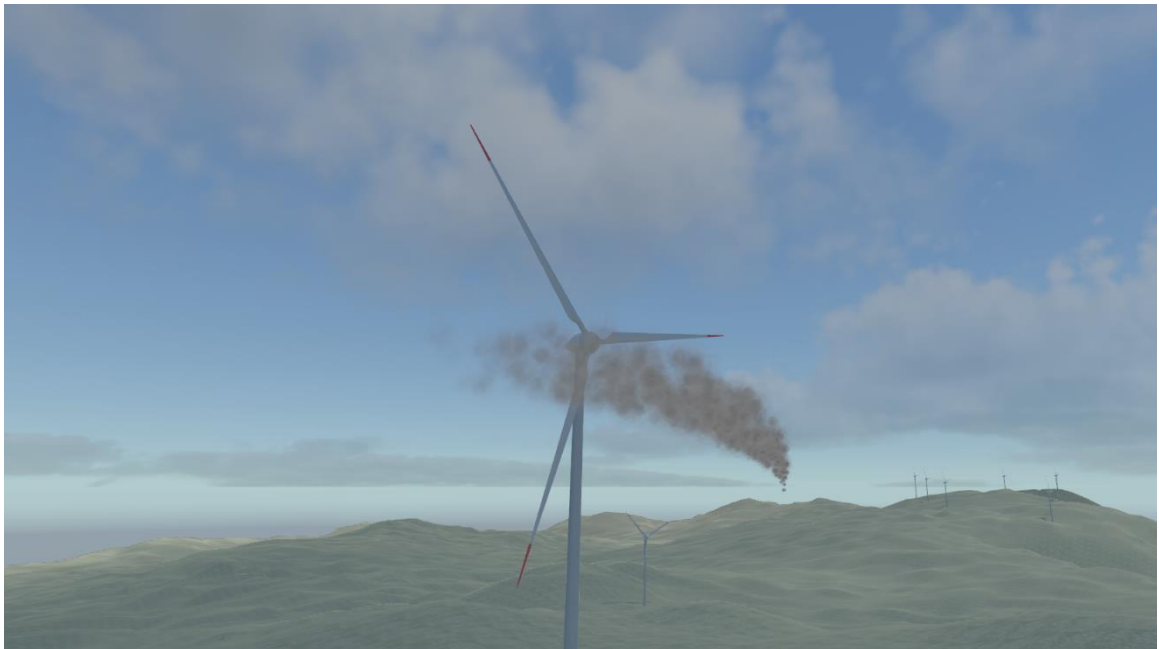


Figure 5.7 Interaction of smoke with a wind generator

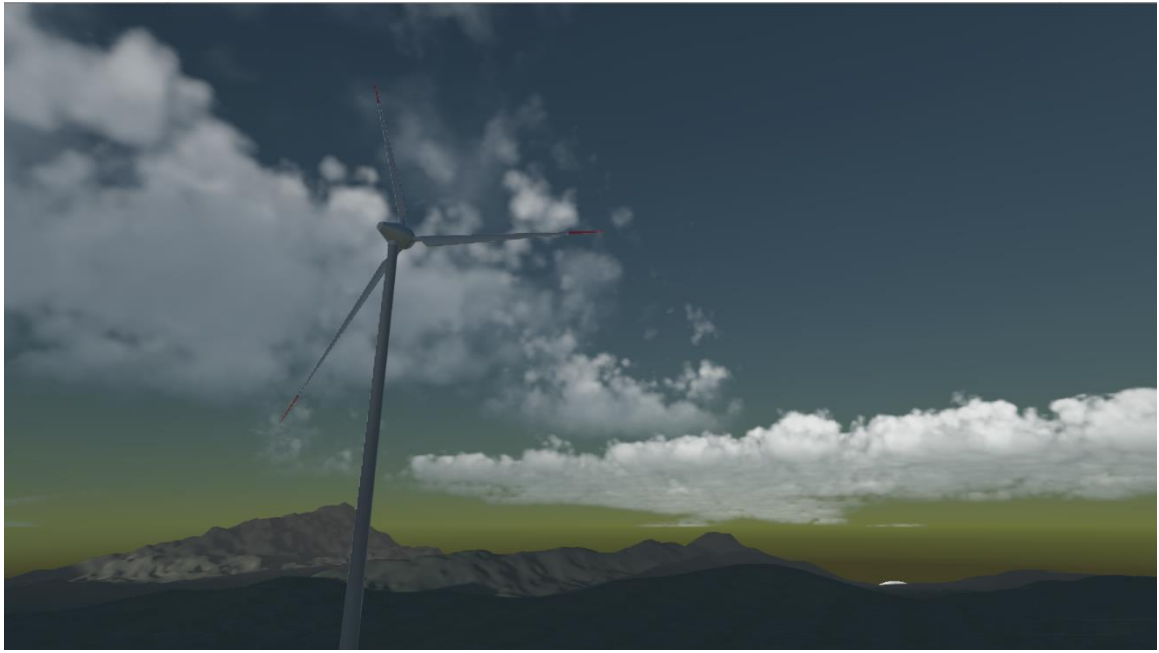


Figure 5.8 Dusklight interacting with the Cumulus clouds



Figure 5.9 Stratocumulus clouds casting a shadow onto the terrain

6. Conclusion

The cloud rendering technique presented in this thesis can produce lots of volumetric clouds under fully dynamic lighting conditions. The technique achieved real-time performance, meaning the render time is under 33 milliseconds i.e. framerate of 30 frames per second, giving us smooth and responsive gameplay. To achieve such fast render time, we had to implement few optimizations and use approximations. This shows how powerful and versatile commercial game engines are, to have the ability to implement cutting-edge techniques while also maintaining visual fidelity. Furthermore, many other people in the industry are beginning to incorporate this technique into their rendering engines, both at game and animation studios.

One optimization that seems very promising is to pre-compute shadows. Since we assume that the direction to the sun is parallel, shadows could be pre-calculated and stored in a look-up table. This look-up table would only need to be updated when the sun has moved. Since we take four additional samples towards the sun for every sample this would greatly reduce the number of samples.

The implementation of the phase function also posed an interesting question: is it worth three times larger performance impact to read the texture with the realistic phase function, or are we satisfied with it being approximated and calculated in the fragment shader?

Although it can impact application's performance, volumetric cloud rendering brings great benefits to the table, such as easily configurable weather, easily configurable clouds, better realism and immersive surroundings. All these attributes are useful to the game designers who are trying to induce emotions in gamers.

Bibliography

- [1] M. Ikits, J. Kniss, A. Lefohn and C. Hansen. *GPU Gems: Volume rendering techniques*.
https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch39.html,
2004. [Accessed: 25/04/2018]
- [2] Patapom. *Real-Time Volumetric Rendering*.
<http://patapom.com/topics/Revision2013/Revision%202013%20-%20Real-time%20Volumetric%20Rendering%20Course%20Notes.pdf>, 2013.
[Accessed: 17/06/2018].
- [3] Í. Quílez, *Raymarching distance fields*,
<http://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm>,
2008. [Accessed 18/05/2018].
- [4] A. Schneider and N. Vos. *The Real-Time Volumetric Cloudscapes of Horizon Zero Dawn*. <https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn>, 2015. [Accessed 10/06/2018].
- [5] A. Schneider. *Nubis: Authoring Real-Time Volumetric Cloudscapes with the Decima Engine*. <https://www.guerrilla-games.com/read/nubis-authoring-real-time-volumetric-cloudscapes-with-the-decima-engine>, 2017. [Accessed: 12/06/2018]
- [6] M. J. Harris and A. Lastra. *Real-Time Cloud Rendering*. Eurographics 2001, 2002.
- [7] R. Olajos. *Real-Time Rendering of Volumetric Clouds*. Master's thesis. Lund University, 2016.
- [8] A. Sachan and M. Seshadri. *Meteoros*. Final project. University of Pennsylvania, 2017.
- [9] D. W. Hahn. *Light Scattering Theory*.
<http://plaza.ufl.edu/dwhahn/Rayleigh%20and%20Mie%20Light%20Scattering.pdf>, 2009. [Accessed: 12/04/2018]
- [10] K. Perlin. *An image synthesizer*. SIGGRAPH Computer Graphics: Volume 19 Issue 3, 1985.
- [11] S. Worley. *A cellular texture basis function*. SIGGRAPH '96, 1996.

Volumetric Atmospheric Effects Rendering

Abstract

In search of a better and more realistic game environment, graphic designers together with game developers have implemented a new way to render clouds. The standard way of rendering clouds consists of skyboxes with flat textures that follow the player in the scene, and it works well in 3D scenes where the camera is expected to be far away from the clouds. But in open world games where the position of the camera cannot be assumed, skyboxes give a static impression and flat textures can give artifacts.

Volumetric clouds are dynamic and look realistic but can also be easily manipulated which is something game designers require. Rendering volumetric clouds is a compute-intensive process which makes it difficult to use in real-time applications.

This thesis presents a way of rendering volumetric clouds in Unity by using precalculated 3D cloud textures and real-time shader calculations.

Keywords: ray marching, volumetric clouds, volumetric shading, real-time rendering

Volumetrijske tehnike u ostvarivanju prikaza atmosferskih učinaka

Sažetak

U potrazi za boljom i vizualno ljepšom scenom u igrama, grafički dizajneri u suradnji sa programerima za igre su implementirali nov način kako iscrtati oblake. Uobičajen način iscrtavanja oblaka se sastoji od pozadine sa plosnatim teksturama koje prate igrača u sceni, taj pristup funkcionira dobro u 3D scenama gdje se očekuje da će kamera biti daleko od oblaka. Ali u igrama otvorenog tipa gdje se pozicija kamere ne može pretpostaviti, pozadina može dati statički dojam, a plosnate teksture mogu dati artefakte.

Volumetrijski oblaci su dinamički i izgledaju realistično te se također mogu lako manipulirati što je privlačno dizajnerima igara. Iscrtavanje volumetrijskih oblaka je računski intenzivan proces što otežava njihovu primjenu kod aplikacija u stvarnom vremenu.

Ovaj rad prikazuje način iscrtavanja volumetrijskih oblaka u Unity razvojnoj okolini koristeći unaprijed izračunate 3D teksture oblaka i sjenčanje u stvarnom vremenu.

Ključne riječi: sjenčanje pretragom zrake, volumetrijski oblaci, volumetrijsko sjenčanje, iscrtavanje u stvarnom vremenu