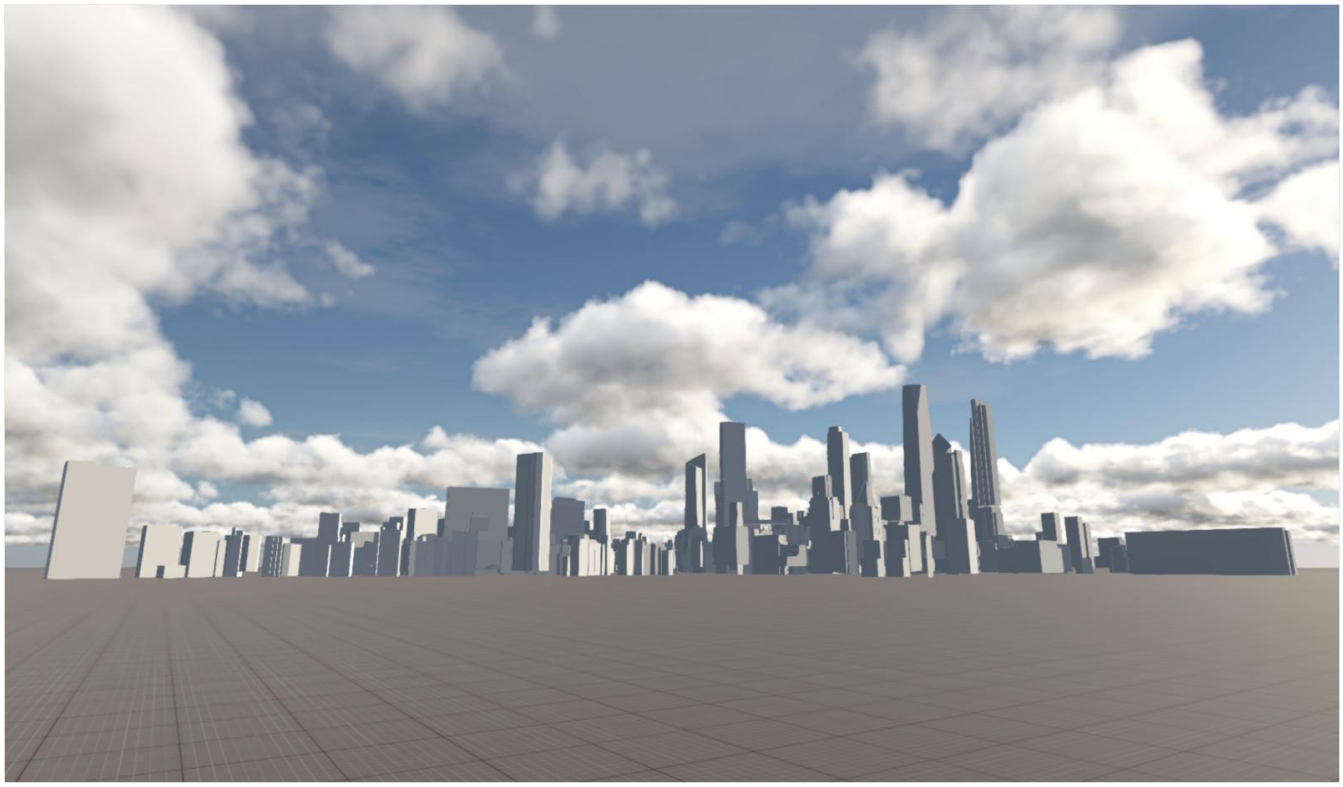

Convincing Cloud Rendering

An Implementation of Real-Time Dynamic Volumetric Clouds in Frostbite

Master's thesis in Computer Science – Computer Systems and Networks

RURIK HÖGFELDT

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Convincing Cloud Rendering - An Implementation of Real-Time Dynamic Volumetric Clouds in Frostbite

Rurik Högfeldt

Supervisor: Erik Sintorn
Department of Computer Science and Engineering
Examiner: Ulf Assarsson
Department of Computer Science and Engineering

Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31 772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden, 2016

Abstract

This thesis describes how real-time realistic and convincing clouds were implemented in the game engine Frostbite. The implementation is focused on rendering dense clouds close to the viewer while still supporting the old system for high altitude clouds. The new technique uses ray marching and a combination of Perlin and Worley noises to render dynamic volumetric clouds. The clouds are projected into a dome to simulate the shape of a planet's atmosphere. The technique has the ability to render from different viewpoints and the clouds can be viewed from both below, inside and above the atmosphere. The final solution is able to render realistic skies with many different cloud shapes at different altitudes in real-time. This with completely dynamic lighting.

Acknowledgements

I would like to thank the entire *Frostbite* team for making me feel welcome and giving me the opportunity to work with them. I would especially want to thank my two supervisors at *Frostbite*, *Sébastien Hillaire* and *Per Einarsson* for their help and guidance throughout the project.

I also want to thank *Marc-Andre Loyer* at *BioWare* for his help and support with the development and implementation.

Finally, I would also like to thank my examinator *Ulf Assarsson* and supervisor *Erik Sintorn* at *Chalmers University of Technology* for their help and support.

Contents

1

Introduction

Real-time rendering of realistic and convincing cloud scenes has always been a desired feature in computer graphics. Realistic and convincing cloud scenes is 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. Many cloud rendering techniques have been developed over the years and are still being researched. Rendering realistic and convincing cloud scenes is still a difficult task as clouds are not only volumetric and dynamic but also requires a complex light transport model.

The cloud system presented in this thesis is inspired by a recent technique developed by Schneider and Vos [AN15]. The main additions and changes to this technique is a different and unified height signal and control. The amount of resources required has also been reduced.

1.1 Motivation

As the computing power increases so does the possibility to use new techniques that previously were only suited for offline rendering to be used in real-time. The area of cloud lighting and 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 scalable enough to be used in real-time rendering in a game context. One common solution is to use impostors or panoramic textures that are applied to the sky. This can produce realistic high definition clouds but it is a very static solution. Using it within a framework featuring dynamic time of day and animations would be very diffi-

cult. These two-dimensional solutions are also not well suited for clouds close to the viewer as they will appear flat and not give a realistic representation of clouds. However these techniques might be suitable for thin layered clouds far away from the viewer.

1.2 Goal

The goal of this thesis was to investigate and implement a technique for rendering realistic and convincing clouds in real-time. The rendering technique should be able to produce a vast amount of clouds that are different in both shape, type and density, while still being dynamic and evolve over time. Although the focus of this technique is on dense clouds close to the viewer. It should be easy to control for an artist via a weather system that can model the clouds shape and behavior. The clouds should also be rendered into a spherical atmosphere which would allow them to bend over the horizon. It is also important that clouds can receive shadows from its environment, both from the planet and from other clouds.

1.2.1 Approach

The approach was to investigate a different solution that can render more interesting and complex cloud scenes. This was be done by studying recent advances in volumetric rendering of clouds and see if and how they can be improved and applied in a game context. The solution was evaluated both by the time it takes to render and the amount of resources required. In order to evaluate how realistic the clouds are actual photographs are used for comparison.

1.3 Report Structure

This paper has the following structure. In Section 2 different cloud types, lighting, their behavior and how they appear are described. Section 3 introduces the reader to previous research of rendering clouds by other authors. In Section 4 the implementation and modeling of these clouds are covered. In Section 5 rendering is described. Section 6 covers optimizations needed for achieving real-time performance. Section 7 presents results from this rendering technique with and without optimizations. This section also offers a visual comparison between in-game rendering and photographs. In Section 8 is a discussion and conclusion of the results and limitations from this implementation. Section 9 covers proposed future work. Finally in Appendix A is additional visual results and in Appendix B is a description of exposed controls.

2

Cloud Physics

Water vapor is invisible to the human eye and it is not until water condensates in air that clouds appear. This section provides a short introduction to different cloud types and how they are named. It also covers behavior, physical properties and simplifications made for modeling clouds. Finally it offers a section about lighting and radiative transfer in participating media.

2.1 Types

Cloud can appear in many different shapes and variations. Most cloud types are named after a combination of its attributes. Attributes that refers to the altitude are *Cirrus* for high altitude and *Alto* for mid altitude. The attribute *Cumulus* is used for clouds that have a puffy appearance. Clouds that have the attribute *Status* appears as layers. The last common attribute is *Nimbus* which is used for clouds with precipitation. In Figure 2.1 common cloud types and their names are displayed.

2.2 Behavior

When modeling clouds it is important to follow their physical behavior for them to be realistic. Rendering clouds without taking physical behavior into account will not yield convincing and realistic results [BNM+08]. 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 form into clouds. Air temperature decreases over altitude and since saturation vapor pressure strongly decreases with temperature, dense

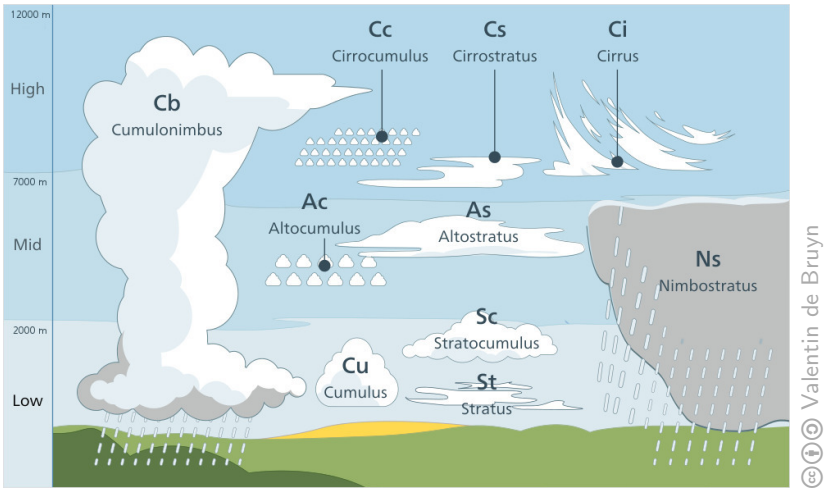

Figure 2.1 – Common cloud types and their names

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 absorbs more and scatter less light [Mh11].

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 of these clouds move in the same wind direction. This makes the behavior quite complex so a few simplifications were done. Some of these simplifications are shown in the list below.

- *Cloud density*: The density inside a cloud increases over altitude but is independent of where in the atmosphere the cloud appears.
- *Wind direction*: Since focus is on low altitude clouds we only take one wind direction into account.
- *Droplet size*: In our cloud model we assume that clouds always have the same droplet size and instead only the density varies.
- *Precipitating clouds*: Instead of modeling precipitating clouds with different droplet sizes we increase the absorption coefficient of these clouds.
- *Atmosphere shape*: We assume that atmosphere can be treated as a perfect sphere instead of an oblate spheroid.

- *Direction to sun*: We assume that the direction to the sun can be treated as parallel within the atmosphere.

2.3 Lighting

This section covers light behaviour when traveling through participating media. Real world clouds do not have a surface that reflects light. Instead light travels through them at which photons interact with particles which may absorb or scatter them, which causes a change in radiance. There are four different ways radiance may change in participating media, these four different ways are described in Figure 2.2. It can be due to absorption, in-scattering, out-scattering or emission [Jar08].

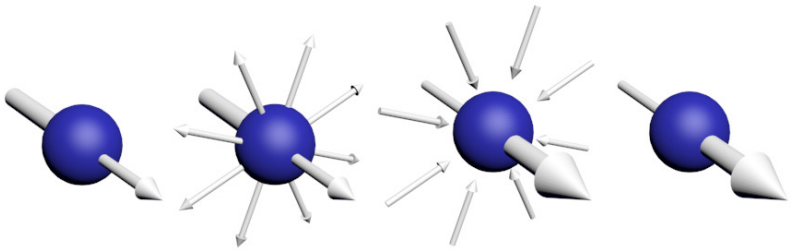

Figure 2.2 – The four ways light can interact with participating media. From left to right, Absorption, Out-scatter, In-scatter and Emission

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 by transforming light into heat. 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.

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

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 $\vec{\omega}$. Many different phase functions exists and are suitable for different types of participating media. The phase function can scatter light uniform in all directions as the isotropic or scatter light differently in forward and backward directions. A phase function for clouds can be very complex as seen in Figure 2.3, which was used in [BNM+08].

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

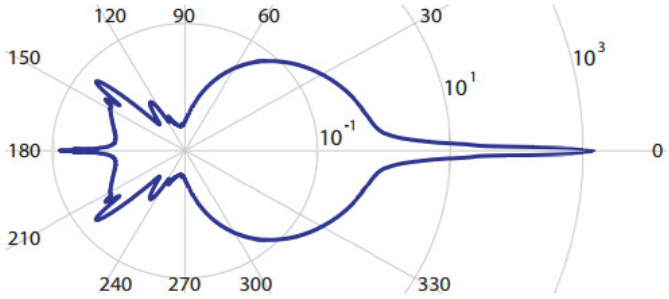

Figure 2.3 – Plot of a Mie phase function used for clouds in [BNM+08]

Clouds are generally white because they scatter light independently of wave length as oppose to atmospheric scattering which scatters blue wave lengths more than others.

2.3.3 Extinction

Extinction coefficient σ_t is the probability that photons 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 photons either being absorbed or out-scattered as described in Equation 2.3.

$$\sigma_t = \sigma_a + \sigma_s \tag{2.3}$$

2.3.4 Transmittance

Transmittance T_r is the amount of photos that travels unobstructed between two points along a straight line. The transmittance can be calculated using *Beer–Lambert's law* as described in Equation 2.4.

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

2.3.5 Emission

Emission is the process of increased radiance due to other forms of energy has transformed into light. Increased radiance because of 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 inside it.

2.3.6 Radiative Transfer Equation

By combining the equations of the four ways light can interact with participating media, it is possible to derive the radiative transfer equation though the law of conservation of energy. The radiative transfer equation is shown in Equation 2.5 and describes the radiance at position x along a ray with direction $\vec{\omega}$ within a participating medium [Cha60].

$$\begin{aligned} L(x, \vec{\omega}) = & T_r(x, x_s)L(x_s, -\vec{\omega}) + \\ & \int_0^s T_r(x, x_t)L_e(x_t, -\vec{\omega})dt + \\ & \int_0^s T_r(x, x_t)\sigma_s(x_t)L_i(x_t, -\vec{\omega})dt \end{aligned} \tag{2.5}$$

3

Related Work

Techniques for rendering clouds are a well-researched area with many recent new breakthroughs and ideas of how to render realistic clouds. Hunagel et al. presented in [HH12] a comprehensive survey on research and development in rendering and lighting of clouds. In this survey the authors compare different techniques and weigh them against each other. Some techniques that are covered are billboards, splatting, volume slicing and ray marching. A table with technique and suitable cloud type is presented, making it easy for the reader to compare.

One recent technique is a cloud system developed by Schneider et al. [AN15] that is going to be used in the game *Horizon Zero Dawn*. By using a combination of Worley and Perlin noise and ray marching, the authors manage to render very realistic cumulus shaped clouds in real-time under dynamic lighting conditions.

Another recent rendering technique for clouds was developed by Egor Ysov [Yus14]. This technique uses pre-computed lighting and particles to render realistic cumulus clouds. But this technique depends on a feature called Pixel Synchronization for providing volume aware blending which is only available on Intel HD graphic cards [Sal13]. Volume aware blending could be implemented using rasterizer ordered views which is a new feature in DirectX 12, but since our implementation must work on all three platforms PC, PlayStation and Xbox this technique is not suitable for our use case.

Bouthors et al. propose in [BNL06] a technique for real-time realistic illumination and shading of stratiform clouds. By using an advanced lighting model were they account for all light paths and preserves anisotropic behavior as well as using a physically based phase function they manage to render realistic clouds in 18-40 fps. This technique is limited to a few cloud types and is mostly suitable for stratiform clouds.

In [BNM+08], Bouthors et al. propose a technique for simulating interactive multiple anisotropic scattering in clouds. With this technique using a similar approach to [BNL06] they manage to also render very realistic lighting of detailed cumulus typed clouds in 2-10 fps.

4

Implementation

This section covers how our new technique was implemented and added to the system. It also covers resources required, how they are used and how they are generated.

4.1 Integration into Frostbite Sky Module

Frostbite has several different sky modules that can be used for rendering skies. Our implementation is focused on a sky module called *Sky Physical*[BS14]. This section provides an overview to the new technique and how it is implemented. First we described how our cloud module was added to the system. The cloud module is responsible for maintaining results from our shader and providing input to it. Then is an short overview of how the shader works.

4.1.1 Cloud Module

A system diagram of the previous technique for rendering clouds before this implementation is shown in Figure 4.1. The clouds were rendered by applying static textures called panoramic and cloud layers to the sky. This technique can produce convincing skies but it is limited to distant clouds because clouds close to the viewer will appear flat.

In order to produce realistic clouds close to the viewer a dynamic cloud rendering technique was needed. The static cloud texture solution was only suitable for distant clouds under static lighting conditions. The new solution should not only produce dynamic clouds but also improve the previous cloud lighting

model by handling dynamic lighting conditions. Therefore a cloud module was developed as shown in Figure 4.2. This module can be added by artists if needed for a scene. The cloud module renders to a texture which is then provided to the sky module. This texture is rendered once for every frame, providing a dynamic solution for rendering clouds. Since the new implementation is focused on rendering clouds close to the viewer the old cloud layer system can still be used for adding high altitude clouds.

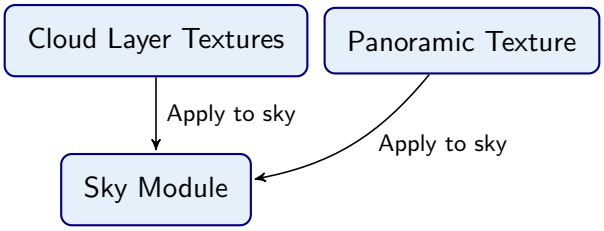

Figure 4.1 – Diagram showing how clouds were rendered before

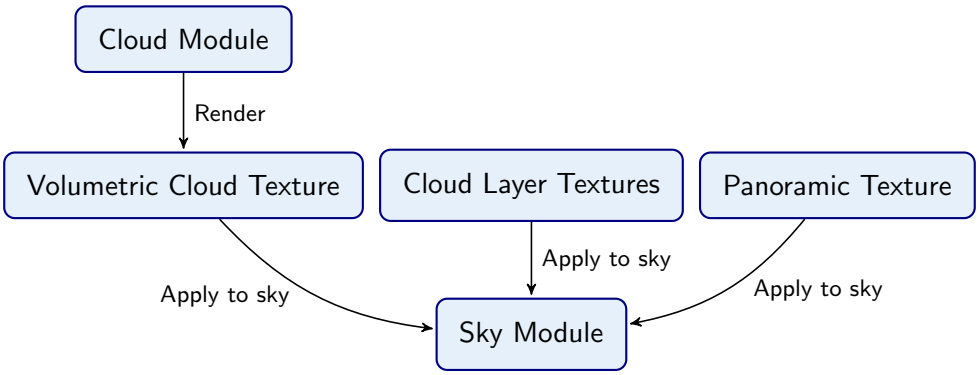

Figure 4.2 – Diagram showing how cloud module was added

4.1.2 Cloud Shader

We also created a single shader to use with our cloud module. This shader is dispatched from our cloud module and uses ray marching together with different noise textures to render volumetric clouds. The following Section 4.2 covers these noises and the shader is covered in detail in Section 5.

4.2 Noise Generation

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.

4.2.1 Perlin Noise

In 1985 Ken Perlin presented a technique for generating natural appearing noise [Per85]. Since then this technique has been widely used generating noise for many natural phonemes including clouds. We generated a tiling three dimensional texture using a program developed Stefan Gustavson [Gus05]. In Figure 4.3 is a three dimensional tiling texture with this noise mapped to a cube.

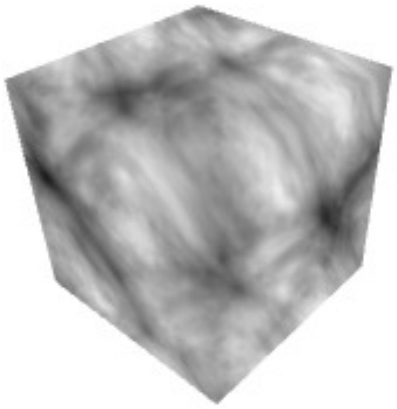

Figure 4.3 – Tiling 3D Perlin noise applied to a cube

4.2.2 Worley Noise

Stewen Worley described a technique for generation cellular noise in [Wor96]. In Figure 4.4 is a cube with three-dimensional tiling Worley noise. We use this noise type to create both wispy and billowing shaped clouds. By inverting Worley noise it is possible to control the appearance between wispy and billowing and vice versa. The algorithm for creating a texture with this noise can be quite simple. A naive approach would be to generate a set of points called feature points and then shade every texel by its distance to the closest feature point. Generating the noise this way would be very slow especially in three dimensions. Therefore the naive approach was 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 exactly one feature point per cell.
3. For each point inside the cuboid color it by the Euclidean distance to the closest feature point. This distance is found by evaluating the feature point inside the surrounding 26 cells and the feature point inside the current cell. By wrapping cells at the edges the texture will be tileable in all three dimensions.

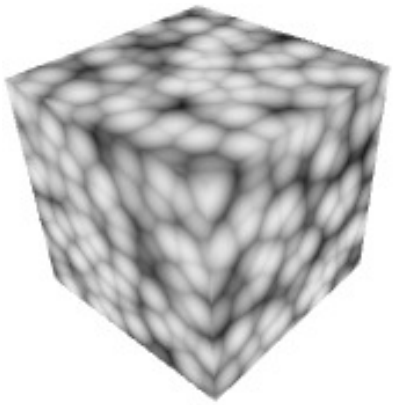

Figure 4.4 – Tiling 3D Worley noise applied to a cube

Since we place exactly one feature point inside each cell this ensures that there will not be any large dark areas which could be the case if feature points were randomly positioned. In Figure 4.4 a cube with size 128^3 and a cell size of 16 is used which yields exactly 8 feature points in any direction in the cube. The result is cellular noise that looks random without having any large dark areas due to feature point being too far apart.

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 the interval 2^n. For example generating Worley noise with 4 octaves and a cell size of 2 as starting frequency makes the following three octaves to be 4, 8, 16.

4.3 Cloud Textures

The noises are pre-generated and stored in two different three dimensional tiling textures as described in Table 4.1. In Figure 4.5 is an example of both the shape and detail texture. The first three dimensional texture is used to create the base shape of the clouds. It has four channels, one with Perlin noise and three with different octaves of Worley. Our clouds will repeat less in the y axis and therefore the size in this axis is smaller in order to reduce texture size. The second three dimensional texture is used for adding details and has three channels with different octaves of Worley noise.

Table 4.1 – Noise textures used to create the cloud shapes

Texture	**Size**	**R**	**G**	**B**	**A**
Shape	128x32x128	Perlin	Worley	Worley	Worley
Detail	32x32x32	Worley	Worley	Worley	-

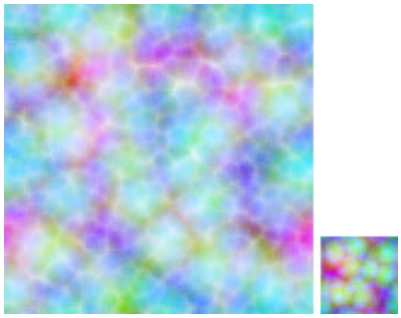

Figure 4.5 – The two cloud textures with channels as described in Tabel 4.1. Left: Shape. Right: Detail

4.4 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. The coverage which is also used as density is controlled by the red channel in this texture. It is also possible to manipulate the coverage by for example modifying it based on the distance to the viewer. Which makes it possible to either increase or decrease clouds presence at the horizon. 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 that controls at which altitude clouds should appear. A value of 0 will yield a start altitude equal to the start of the atmosphere and a value of 1 makes the cloud appear at maximum cloud layer altitude. These two maximum values are exposed as settings in our module and can easily be controlled at run-time. In Table 4.2 is a description of the different channels in the weather texture.

In Figure 4.6 is an example of a weather texture used during the implementation. The overall yellow tone is due to the coverage and height having similar values. The blue which controls altitude is set to zero except for four different clouds. Therefore clouds using this weather texture will appear at five different altitudes.

Table 4.2 – Channel description of the weather texture

R	**G**	**B**	**A**
Coverage	Height	Altitude	-

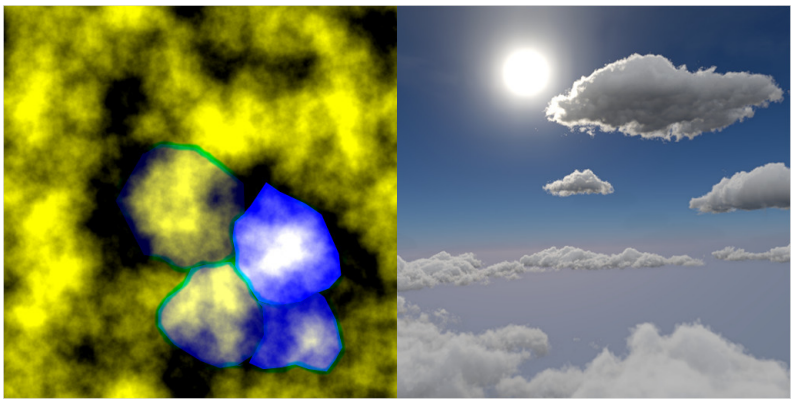

Figure 4.6 – Example of a weather textures and resulting clouds. Left: Weather texture. Right: In-game render using this weather texture.

4.4.1 Precipitation

Larger water droplets absorbs more light and have a higher presence at the base of clouds. Since we assume uniform droplet size it is not possible to achieve this by only changing the absorption coefficient. This is even more complicated since density also increases over altitude which makes the bottom part of a

cloud absorb less than the top, which is opposite of what is expected. Therefore our rain clouds are darkened by a gradient which is multiplied with light contribution. With this simplification all clouds can have the same absorption coefficient while the technique is still being able to produce rain and non-rain clouds.

4.4.2 Wind

In the real world clouds can move in different directions at different altitudes. Our technique is limited to a single wind direction applied as an offset to the weather texture. But since we still support the old cloud layer system each layer can have wind applied in different directions. It is therefore possible to have high altitude clouds move in a different direction than low altitude clouds.

4.5 Shape Definition

This section covers how cloud shapes are defined from noises described in Section 4.2. The base cloud shapes are defined by the first three-dimensional texture. The Worley noise channels are summed and multiplied with the Perlin noise channel.

4.5.1 Density Modifiers

All clouds are modeled using the same approach and the density is independent of where in the atmosphere it appears. First the coverage value from weather texture in 4.4 is used as initial density. Then a height signal covered in Section 5.4 is applied. This height signal lowers density at both the top and the bottom at the cloud. Then the two three-dimensional textures are used to erode density from the height signal. 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 calculations are done must be in 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 in order to increase robustness and make lighting more balanced.

4.5.2 Detailed Edges

Defining a cloud shape from only the shape 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 detail

texture with Worley noise. By inverting Worley noise it is possible to change between wispy and billowing edges or use a combination of both.

The amount of erosion is determined by the density, lower densities are eroded more. Strength of erosion is implemented as a threshold value which only erodes densities lower than this value. Algorithm 4.1 describes the order of how the density is calculated.

Algorithm 4.1 – How density is calculated

```
weather = getWeather(position.xz)
density = weather.r
density *= HeightSignal(weather.gb, position)
density *= getCloudShape(position)
density -= getCloudDetail(position)
density *= HeightGradient(weather.gb, position)
```

5

Rendering

This section describes how clouds are rendered and lit according to special density distribution presented in previous section. First we describe how clouds are rendered using ray marching from views below the atmosphere. We then describe how this can easily be extended into rendering from views inside the atmosphere and from space. This section also covers how a spherical atmosphere was achieved. It then describes how clouds can be rendered with any height at any altitude using our improved height signal. We then cover how lighting is applied and how shadows are added.

5.1 Ray Marching

Ray marching is the volumetric rendering technique that we use for rendering clouds. A scene is rendered using this technique by for each pixel march along a ray and evaluating density, lighting and shadow at each sample point along that ray. More sample points yields better results but in turn is more expensive. Therefore it is necessary to weigh the step count and step length against each other, in order to get desired result.

In our solution we use a step size which is the draw distance divided by the number of steps. Draw distance is the minimum value of the atmosphere depth and a artist controlled cutt-off distance. The number of steps is determined by both quality option as well as the depth of the atmosphere. This makes sure that even if the atmosphere would be scaled to be very small or very large an appropriate step count will always be used. As seen in Figure 5.1 the atmosphere depth is larger near the horizon but the step count is kept constant. This has the effect that clouds close to the viewer has a smaller step size which yields better quality. Algorithm 5.1 shows how ray marching

is performed when viewed from planet surface.

Algorithm 5.1 – Ray marching through atmosphere

```
entry = RaySphereIntersect(atmosphereStart, directionFromEye)
exit = RaySphereIntersect(atmosphereEnd, directionFromEye)
stepLength = Distance(entry, exit) / steps
stepOffset = random[0,1]
t = stepLength * stepOffset
for step=0; step < steps; ++step
    position = entry + directionFromEye * t
    density = GetDensity(position)
    EvaluateLight(density)
    t += stepLength
```

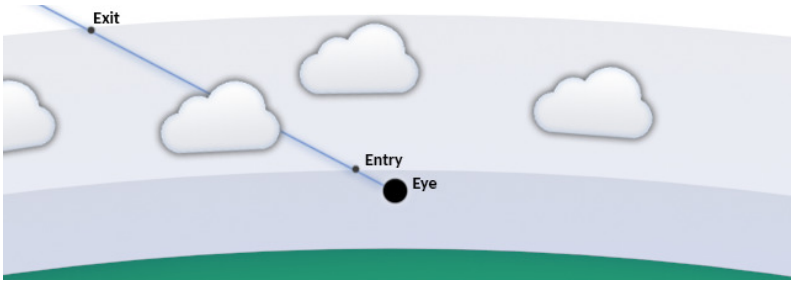

Figure 5.1 – How atmosphere depth depends on view direction

5.2 Render From Inside Atmosphere and Space

Algorithm 5.1 can easily be extended to render clouds from views within the atmosphere and from space. When we render from space view we use exit point in Figure 5.1 as origin and march into the atmosphere. It is not possible to calculate atmosphere depth by using the distance between entry and exit points. This is because when using space view not all rays will intersect with both the inner and outer bounds of the atmosphere. This is solved by using a fixed atmosphere depth for rays that only intersected with the outer bound of the atmosphere.

When rendering from within the atmosphere we use the eye position as origin instead of using the entry point. Atmosphere depth is calculated as the distance from the eye to the exit point and with a maximum of cutt-off distance. Automatic change of ray marching algorithm is handled by Cloud Module.

5.3 Dome Projection

Dome projection is implemented by using ray-sphere intersection as described in Algorithm 5.2. This algorithm calculates intersecting points of a ray and a perfect sphere. Earth's atmosphere is shaped as an oblate spheroid e.i. the radius is larger at the equator than at the poles, but in order to simplify calculations we treat the atmosphere as a perfect sphere instead. By using the position returned by this algorithm as entry point clouds will naturally bend over the horizon.

Algorithm 5.2 – Ray sphere intersection

```
a = dot(direction, direction) * 2
b = dot(direction, startPosition) * 2
c = dot(start, start)
discriminant = b * b - 2 * a *
    (c - atmosphereRadius * atmosphereRadius)
t = max(0, (-b + sqrt(discriminant)) / a)
intersection = cameraPosition + direction * t
```

5.4 Height Signal

The height signal of the clouds is implemented as an parabola function as described in Equation 5.1, where x is the altitude in atmosphere, a is cloud starting altitude, h is cloud height. This function controls both the altitude at which the cloud appears and its actual height. The height signal function has two roots, one at the starting altitude of the cloud and the other root at the cloud height. The global maximum value of the height signal is scaled to always be one. By using this parabola function clouds naturally get less density at its top and bottom edges.

$$\underbrace{(x-a)}_{\text{1st root}} \cdot \underbrace{(x-a-h)}_{\text{2nd root}} \cdot \underbrace{\frac{\text{-4}}{h^2}}_{\text{Scale}} \tag{5.1}$$

In Figure 5.2 is a graph showing how the density varies with altitude. In this graph starting altitude a_s is 1 and height h is set to 2, resulting in a density larger than 0 for altitudes between 1 and 3. Algorithm 5.3 shows how this height signal is implemented in the shader.

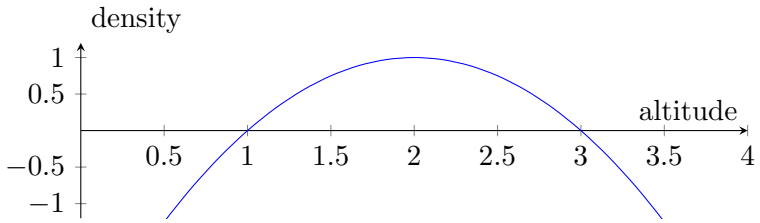

Figure 5.2 – Heigth signal with $a_s = 1$ and $h = 2$

Algorithm 5.3 – Height signal implementation

```
oneOverHeight = 1 / height
altitudeDiff = altitude - altitudeStart
heightSignal = altitudeDiff * (altitudeDiff - height)
    * oneOverHeight * oneOverHeight * -4
```

5.5 Lighting

Lighting is evaulated for every sample that returned a density larger than zero when ray marching through the atmosphere. Our lighting model is a simplification of the radiative transfer function since we do not take emission into account. We also do not account for surfaces behind our clouds. The remaining part of the radiative transfer function that needs to be solved is shown in Equation 5.2.

$$L(x, \vec{\omega}) = \int_0^s T_r(x, x_t) \underbrace{\sigma_s(x_t) L_i(x_t, \text{-}\vec{\omega}) dt}_{\text{S}} \tag{5.2}$$

Transmittance follows the properties of *Beer–Lambert's law* as described in Equation 5.3. By using this property the scattered light and transmittance can be calculated using analytical integration as shown in Algorithm 5.4. This is using a scattering integration method proposed by Hillaire [Sé15] and is more stable for high scattering values than previous integration techniques.

$$\begin{aligned} T_r(x_0, x_1) &= e^{\text{-}\int_{x_0}^{x_1} \sigma_t(x) dx} \\ T_r(x_0, x_2) &= T_r(x_0, x_1) \cdot T_r(x_1, x_2) \end{aligned} \tag{5.3}$$

Algorithm 5.4 – Analytical integration

```
sampleSigmaS = sigmaScattering * density
sampleSigmaE = sigmaExtinction * density
ambient = gradient * precipitation * globalAmbientColor
S = (evaluateLight(directionToSun, position) * phase +
    ambient) * sampleSigmaS
Tr = exp(-sampleSigmaE * stepSize)
    /*
        Analytical integration of light/transmittance
        between the steps
    */
Sint = (S - S * Tr) / sampleSigmaE

scatteredLight += transmittance * Sint
transmittance *= Tr
```

5.5.1 Phase Function

A phase function describes the angular distribution of scattered light. The phase functions is responsible for cloud lighting effects such as silver lining, fogbow and glory. In our implementation we use the Henyey-Greenstein [HG41] phase function described in Equation 5.4. 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. This forward peak is the strongest optical phenomena found in clouds. Another phase function we considered using was one presented by Cornette-Shank [CS92] and is described in Equation 5.5. This phase function is also well suited for clouds but is more time consuming to calculate [FZZZ14].

Both the Henyey-Greenstein and the Cornette-Shank phase function does not produce back-scattered optical phenomenas as the Mie phase function does. A comparison between the Henyey-Greenstein and the Mie phase function is described in Figure 5.3. We did not use the Mie phase function due to its oscillating behavior which is undesired because it could cause banding artifacts.

$$p_{HG}(\theta) = \frac{1 - g^2}{4\pi \cdot (1 + g^2 - 2g \cdot cos(\theta))^{1.5}} \tag{5.4}$$

$$p_{CS}(\theta) = \frac{3}{2}\frac{(1 - g^2)}{(2 + g^2)}\frac{(1 + cos^2(\theta))}{(1 + g^2 - 2g \cdot cos(\theta))^{1.5}} \tag{5.5}$$

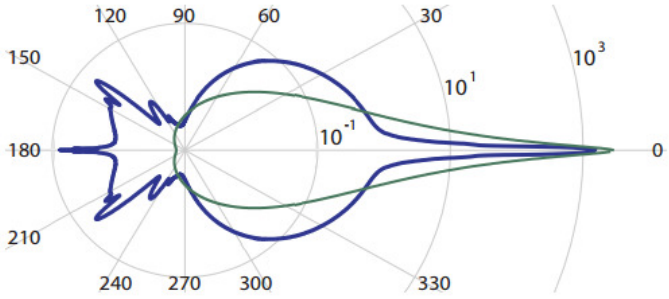

Figure 5.3 – Plot of phase functions for light scattering in clouds [BNM+08]. Blue: Mie, Green: Henyey-Greenstein with $g = 0.99$

5.5.2 Shadows

Self- and inter-cloud shadowing is implemented by for each step inside a cloud also step towards the sun as shown in Figure 5.4. The self- and inter-cloud shadowing effect is expensive although necessary to convey a sense of depth. Marching towards the sun has a great impact on performance since for every step an extra number of steps has to be taken. In our implementation four steps towards the sun are taken at exponentially increasing steps size. This is because we want most of the contribution to be from the cloud itself while still receiving shadows from other clouds. In Figure 5.5 is an in-game rendered image showing clouds with both self- and inter-cloud shadowing.

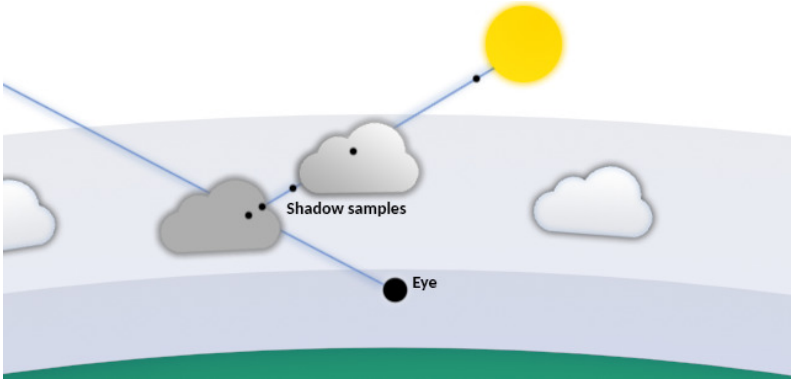

Figure 5.4 – Exponentially increasing step size taken towards the sun

Figure 5.5 – In-game render showing inter cloud shadowing

5.5.3 Ambient

Ambient light contribution is added using a linear gradient from the clouds bottom to the top and increases with altitude. The strength as well as color of the ambient contribution is controlled by the Cloud Module. How ambient contribution is added is described in Algorithm 5.4.

6

Optimization

Several optimizations has been implemented in order for this cloud rendering technique to achieve real-time performance. This section covers most of these different optimizations.

Render Target Resolution
Decreasing the render targets resolution to half of the original resolution is one optimization that was necessary for the rendering technique to achieve real-time performance. Using an even lower resolution may provide an even greater performance boost at the cost of a blurrier result.

Shape Texture Resolution
The first implementation had the same resolution in all axises for the shape texture. But since this texture is repeated more in x and z than y. We reduce the amount of resources required for this texture by using a quarter of the resolution for the y axis.

Temporal Upsampling
By using temporal upsampling it is possible to take fewer steps when marching without reducing visual quality of the clouds. How temporal upsampling was implemented is covered later in this section.

Early Exit
It is possible to early exit in the shader if the transmittance is either high or low as described later in this section. In the rendered images in Section 7.3 with visual results marching was terminated when the transmittance was less than 0.01.

6.1 Temporal Upsampling

This section describes how temporal upsampling was implemented and how the technique improved the visual quality. In Figure 6.1 is a scene without temporal upsampling and in Figure 6.2 is the same scene rendered with temporal upsampling. Both figures are rendered using the same step count and length, the only difference is the temporal upsampling. Notice how the banding artifacts disappears when temporal upsampling is enabled. This is because multiple frames have been blended together and each starting step when ray marching was different. The different steps for temporal upsampling is described below.

1. Offset the starting step in the ray marching algorithm by multipling the step length with a value in the range [0,1], as described in Algorithm 5.1. This value is provided by the cloud module and is calculated as a Van der Corput sequence.

2. Calculate world position of P from clip space position using current inverse view projection matrix.

$$P_{wp} = P_{cs} \cdot (VP)^{-1}$$

3. Calculate, P'_{cs}, which is position of P in the previous frame. This is done by multiplying P_{wp} with view projection matrix from previous frame.

$$P'_{cs} = P_{wp} \cdot (VP)$$

4. Sample previous frame at position P'.

$$C_{P'} = texture(P')$$

5. Blend current frame with previous frame using a blending factor α, which is *TemporalAlpha* in AppendixB. Low values of α results in less banding but takes more time to converge. We use 5% as the default blending value. If the sample is outside of the screen we set α to 1, which results in only the current frame being rendered. In order to hide banding when we cannot blend with the previous frame we increase the step count for this pixel.

$$C_P \cdot \alpha + C_{P'} \cdot (1 - \alpha)$$

Algorithm 6.1 – Temporal upsampling

```
clipSpacePos = computeClipPos(screenPos)
camtoWorldPos = (clipSpacePos, 1) * invViewProjectionMatrix
camtoWorldPos /= camtoWorldPos.w
pPrime = camtoWorldPos * ViewProjectionMatrix
pPrime /= pPrime.w
screenPos = computeScreenPos(pPrime)
isOut = any(abs(screenPos.xy - 0.5) > 0.5)
current = scatteredLight, transmittance.x
result = isOut ? current : current * alpha
    + prevoius * (1 - alpha)
```

Figure 6.1 – Without temporal upsampling

6.2 Early Exit

Performing as few calculations as possible is key to performance. This section describes how we use early exit points in our shader to improve performance.

6.2.1 Low Transmittance

While marching through the atmosphere every sample with a non-zero density will reduce the transmittance, which is loss of energy. Therefore as transmittance gets lower new samples will contribute less to the final result. We stop marching when transmittance is less than 0.01. This provided a great performance increase without noticeable visual artifacts.

Figure 6.2 – With temporal upsampling

Figure 6.3 – Red parts showing when early exit was done due to low transmittance

6.2.2 High Transmittance

When transmittance is high it means that few samples along this ray had a density larger than zero. By using the position calculated for temporal upsampling we can sample transmittance from the previous result and check whether there was a cloud in this direction. If transmittance is less than 1.0 there is a cloud in this direction, but due to wind we cannot assume that a direction with transmittance of 1.0 has not any clouds. Always exiting on high transmittance would not render all clouds if they move into a region where previous frame had high transmittance. Instead we early exit before any

marching is done if transmittance in previous frame is 1.0 and then force this pixel to be marched in the following frame. This could be done automatically using a checkered tile pattern instead to avoid one frame being a lot more expensive than the other.

Figure 6.4 – Red parts showing when early exit was done due to high transmittance

7

Results

This section presents in detail our results from this implementation with resources required, run-time performance and finally a visual comparison. These visual results are focused on clouds rendered from the planets surface. Additional visual results from inside the atmosphere and from space are presented in Appendix A.

7.1 Resources

Table 7.1 shows the amount of resources that this technique requires. The resources required for render tagets is based on using half resolution of 1920x1080 while rendering.

Table 7.1 – Resources required

Texture	**Format**	**Size**	**Memory**
Detail	RGB8	32x32x32	0.1MB
Weather	RGB8	1024x1024	3.1MB
Shape	RGBA8	128x32x128	2.1MB
RenderTarget A	RGBA16	960x540	4.1MB
RenderTarget B	RGBA16	960x540	4.1MB
Total			13.5MB

7.2 Performance

This section describes run-time performance measured on a PC using a built-in performance diagnostics tool. The hardware configuration used during performance evaluation is shown in Table 7.2.

Table 7.2 – Hardware configuration used when evaluating performance

Hardware	**Release year**
XFX Radeon HD7870	2012
Intel Xeon E5-1650	2012

Most computations are performed in the pixel shader on the GPU. Execution times are measured on the GPU using a built-in diagnostics tool. The environment used when measuring was an empty level which only had clouds in it. In Figure 7.1 is execution time shown when maximum draw distance has a constant value of 20 000m and view direction is upwards. In Figure 7.2 is the with maximum draw distance altered and atmosphere depth has a constant value of 2 000m and view direction is towards the horizon for a comparison of how rendering distance affects run-time performance. When coverage is high almost the entire sky is covered by 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 on high transmittance.

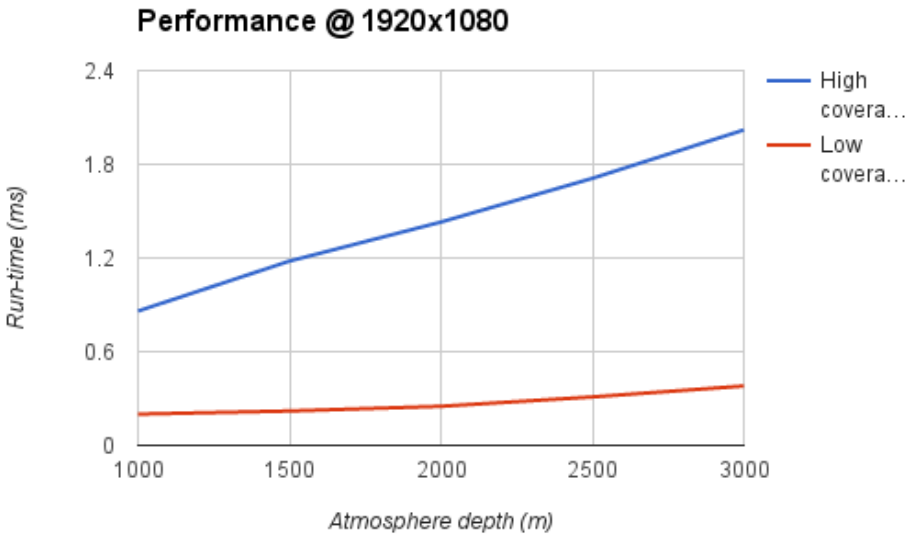

Figure 7.1 – Run-time performance as function of atmosphere depth.

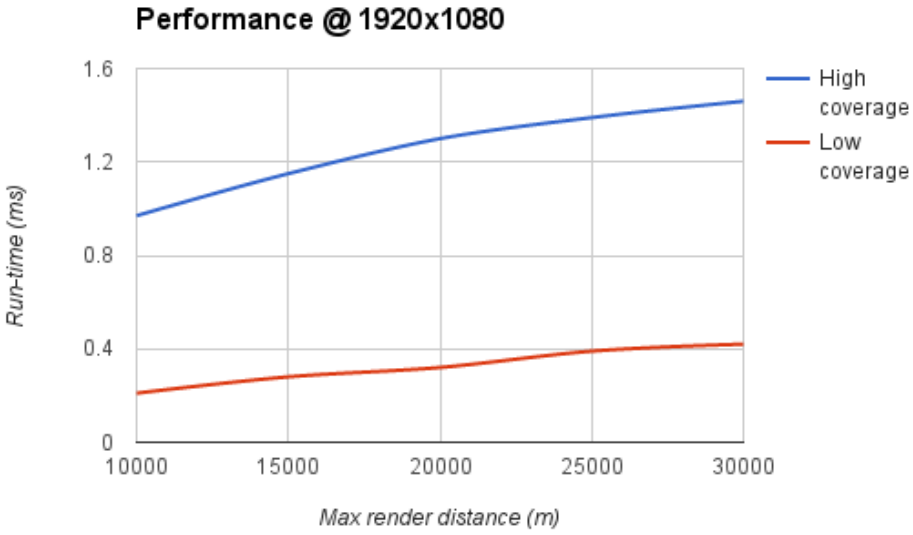

Figure 7.2 – Run-time performance as function of max render distance.

7.3 Visual Results

In this section we present visual results from our cloud rendering technique together with photographs for comparison.

Figure 7.3 – Yellow sunset photograph

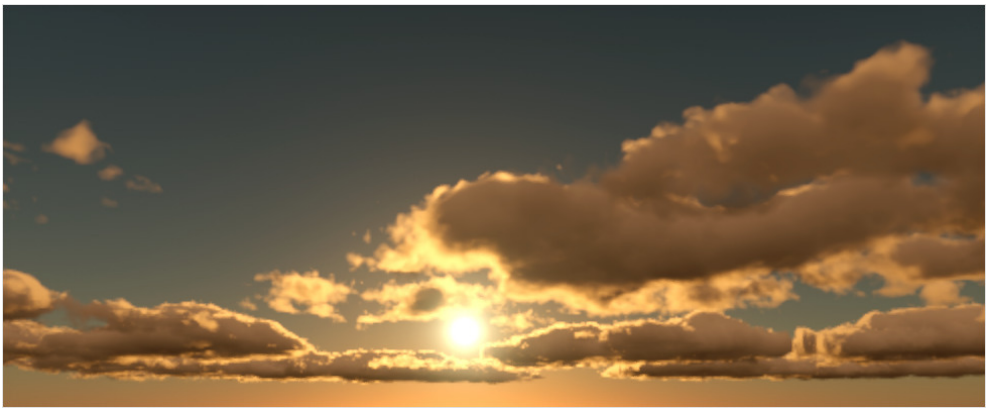

Figure 7.4 – Yellow sunset in-game render

Figure 7.5 – Cumulus clouds in-game render

Figure 7.6 – Cumulus clouds photograph

8

Discussion and Conclusion

The cloud rendering technique we have presented can produce lots of clouds under fully dynamic lighting conditions. The technique achieved real-time performance through several optimizations and can be used in games as seen in Figure A.6, where it was added to *Battlefield 4*. Although the technique is not yet production ready as it missing important features like god rays, reflection views and the clouds does not cast shadows on the environment. In order to implement these features the technique will most likely need additional optimizations. During the implementation we introduced a distance field together with the weather texture that would store the distance to the closest cloud. This distance field was pre-generated using the midpoint circle algorithm to find the closest red pixel in the weather texture and store this inside a different texture. The distance field increase run-time performance in some cases but in others it got worse and was therefore removed.

One optimization that seems very promising but has not been implemented yet 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 weather texture shown in Figure 4.6 has an overall yellow tone due to red and green having similar values. One optimization to reduce the size of this texture could be to remove one of these channels and use the same for coverage and height. This would not allow scenes to have both low and high

clouds with high density, but if it might increase run-time performance.

The Henyey-Greenstein phase function provides a strong forward peek which is necessary for a silver lining effect although its approximation. A Mie phase function for cumulus clouds were generated using *MiePlot* and stored in a texture for use in the shader. Our results from this were an oscillating behaviour from back-scattered light and reduced run-time performance Therefore we instead use Henyey-Greenstein, but using Mie phase function would be a better solution for providing more physically accurate result.

The Mie phase function was tested by generating it using *MiePlot* [Lav45] and storing it in a look-up table which was then used in the shader. This gave promising results but due to the Mie phase functions oscillating behaviour banding occurred. This was most visible when viewing along the sun direction.

8.1 Limitations

This cloud rendering does not support clouds that overlap in altitude. This is because the weather texture only stores coverage, height and starting altitude. By using multiple weather textures this limitation could be resolved, but it would reduce run-time performance.

The phase function we use produces a strong forward peak but does not produce back-scattering effects as glory and fogbow. We also only take sun and ambient light into account and not other light sources.

8.2 Ethical Aspects

This cloud rendering technique can produce clouds for many different weather situations and in a game context changing the weather could affect both the mood and behavior of the players. A more general ethical aspect related computer graphics is photo and video manipulation and falsification.

9

Future work

In our future work we will optimize the algorithm and implement currently missing features such as render into reflection views and cast shadows on the environment. The clouds should also be affected by aerial perspective and fog to allow for a smoother transition into the horizon. Another part that is left for future work is to implement the cloud rendering technique on other platforms such as Playstation and Xbox.

References

[AN15] Schneider Andrew and Vos Nathan. The real-time volumetric cloudscapes of horizon zero dawn. In *SIGGRAPH*, aug 2015.

[BNL06] Antoine Bouthors, Fabrice Neyret, and Sylvain Lefebvre. Real-time realistic illumination and shading of stratiform clouds. In *Eurographics Workshop on Natural Phenomena*, sep 2006.

[BNM+08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 173–182, New York, NY, USA, 2008. ACM.

[BS14] Gustav Bodare and Edvard Sandberg. *Efficient and Dynamic Atmospheric Scattering.* Chalmers University of Technology, http://publications.lib.chalmers.se/records/fulltext/203057/203057.pdf, 2014.

[Cha60] Subrahmanyan Chandrasekhar. *Radiative Transfer.* Dover Publications Inc, 1960.

[CS92] William M. Cornette and Joseph G. Shanks. Physically reasonable analytic expression for the single-scattering phase function. *Appl. Opt.*, 31(16):3152–3160, Jun 1992.

[FZZZ14] Xiao-Lei Fan, Li-Min Zhang, Bing-Qiang Zhang, and Yuan Zha. Real-time rendering of dynamic clouds. *JOURNAL OF COMPUTERS*, 25(3), 2014.

[Gus05] Stefan Gustavson. *Simplex noise demystified.* Linköping University, http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf, 2005.

[HG41] L. G. Henyey and J. L. Greenstein. Diffuse radiation in the Galaxy. *Astrophys Journal*, 93:70–83, January 1941.

[HH12] Roland Hufnagel and Martin Held. A survey of cloud lighting and rendering techniques. In *Journal of WSCG 20, 3*, pages 205–216, 2012.

[Jar08] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

[Lav45] Philip Laven. *MiePlot*. A computer program for scattering of light from a sphere using Mie theory & the Debye series, http://www.philiplaven.com/mieplot.htm, v4.5.

[Mh11] Marco KOK Mang-hin. *Colours of Clouds*. Hong Kong Observatory, http://www.weather.gov.hk/education/edu06nature/ele_cloudcolours_e.htm, 2011.

[Per85] Ken Perlin. An image synthesizer. In *ACM SIGGRAPH Computer Graphics: Volume 19 Issue 3*, Jul 1985.

[Sal13] Marco Salvi. *Pixel Synchronization*. Advanced Rendering Technology Intel - San Francisco, http://advances.realtimerendering.com/s2013/2013-07-23-SIGGRAPH-PixelSync.pdf, 2013.

[Sé15] Hillaire Sébastien. Physically-based & unified volumetric rendering in frostbite. In *SIGGRAPH*, aug 2015.

[Wor96] Stewen Worley. A cellular texture basis function. In *Proceedings of the 23rd annual conference on computer graphics and interactive techniques.*, page 291–294, 1996.

[Yus14] Egor Yusov. High-performance rendering of realistic cumulus clouds using pre-computed lighting. In *Proceedings of High-Performance Graphics*, 2014.

A

Additional Visual Results

Figure A.1 – Sunset with very high start multiplier

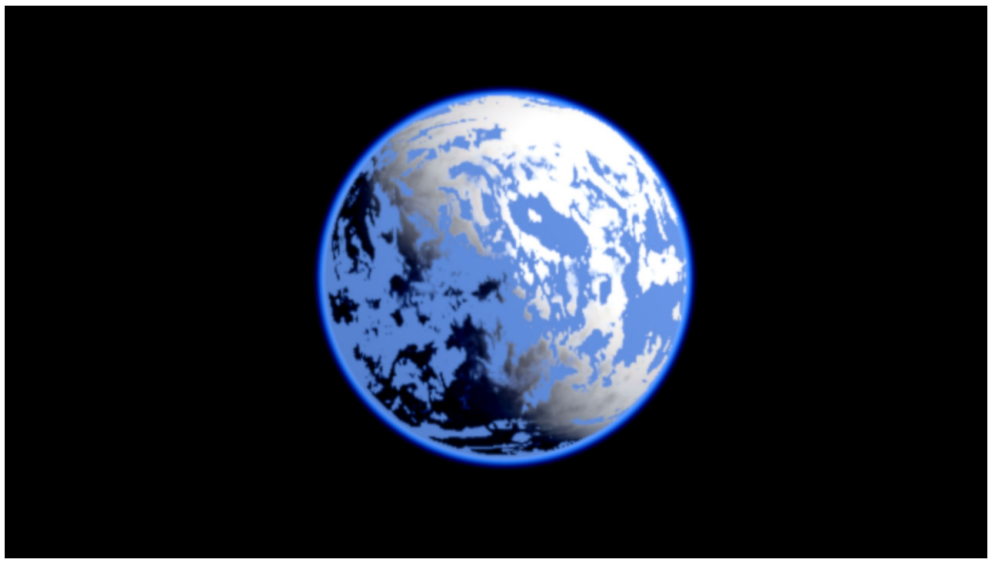

Figure A.2 – Clouds viewed from space

Figure A.3 – Clouds viewed from inside atmosphere at day

Figure A.4 – Sunset viewed from inside atmosphere

Figure A.5 – Sunset viewed from inside atmosphere

Figure A.6 – Cumulus clouds in Battlefield 4

B

Artist Controls

This section lists controls with their names and description that were added to our new cloud module for modifying clouds. The controls are grouped together in sections by what they control. All these controls can be dynamically changed during run-time. Controls for wind and sky are not listed since they are not part of this module. In Figure B.1 is the editor *FrostEd* with different modules and exposed controls for the cloud module displayed.

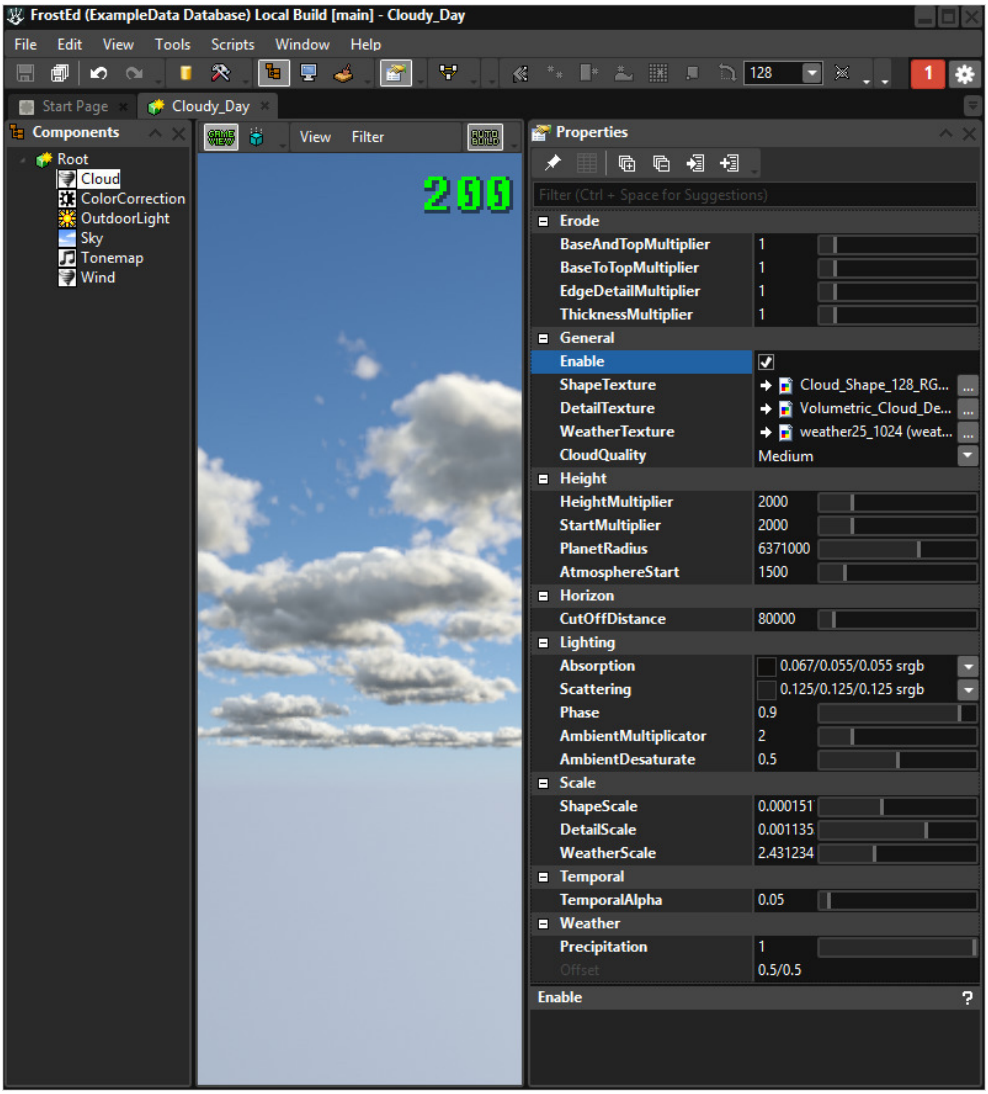

Figure B.1 – Editor showing selected Cloud module with exposed controls

Erode

- **BaseAndTopMultiplier** Modifies strength of a parabola function used as height signal. This is the extrema value of this function.
- **BaseToTopMultiplier** Modifies strength of the linear height gradient.
- **EdgeDetailThreshold** Threshold for specifying at which density details should be added.
- **ThicknessMultiplier** Increases density of the cloud shape before erosion.

General

- **Enable** Enables the cloud module.
- **ShapeTexture** 3D Shape texture.
- **DetailTexture** 3D Detail texture.
- **WeatherTexture** 2D Weather texture.
- **CloudQuality** This is a quality option that allows for a trade-off between performance and visual appearance. It adjusts how many samples are taken when rendering.

Height

- **HeightMultiplier** Specifies the maximum cloud height in meters.
- **StartMultiplier** Specifies the maximum starting altitude of a cloud in meters.
- **PlanetRadius** Radius of the planet in meters.
- **AtmosphereStart** Starting distance of the atmosphere from the planet's surface.

Horizon

- **CutOffDistance** Maximum distance from camera that clouds are rendered in.

Lighting

- **Absorption** Color and strength of absorption coefficient.
- **Scattering** Color and strength of scattering coefficient.
- **Phase** Paramater g in the Henyey Greenstein phase function.
- **AmbientMulitplicator** Strength of ambient contribution.
- **AmbientDesaturate** Modifies the saturation of ambient color.

Scale

- **ShapeScale** Scaling value for shape texture.
- **DetailScale** Scaling value for detail texture.
- **WeatherScale** Scaling value for weather texture.

Temporal

- **TemporalAlpha** Blending value for temporal upsampling between current and previous frame.

Weather

- **Precipitation** Adds preciptation effect to clouds.
- **Offset** Offsets the weather texture.