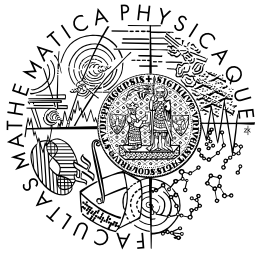


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Koblížek

Procedurally Generated Volumetric Cloudscapes for Unity

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Martin Kahoun
Study programme: Computer Science
Study branch: General Computer Science

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank Mgr. Martin Kahoun and Doc. Ing. Jaroslav Křivánek, Phd. for their advice and help with my thesis.

Title: Procedurally Generated Volumetric Cloudscapes for Unity

Author: Jan Koblížek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Martin Kahoun, Department of Software and Computer Science Education

Abstract:

The traditional approach to cloud rendering in computer games is based on static skyboxes or a set of static textures. Volumetric clouds used to be too computationally expensive, but with advances in GPU performance, they were successfully used in recent gaming titles.

This thesis presents the implementation of real-time volumetric clouds for the Unity game engine. Clouds are described by multiple textures (both 3-dimensional and 2-dimensional) and rendered using a ray marching algorithm.

The resulting implementation allows three types of low altitude clouds – cumulus, stratocumulus and stratus. The user can seamlessly transition between different coverages, times of the day, and animate clouds based of the speed and direction of the wind. Clouds support advanced lighting effects such as casting soft shadows and sun shafts.

Keywords: clouds, volumetric raymarching, real-time rendering, Unity (game engine)

Contents

Introduction

Just as in real life, in video games, the sky often covers close to half of the view and plays an important role in setting up the general mood of the environment. Light sky with a minimal amount of clouds can give the scene a more positive feeling, while dark overcast sky evokes a feeling of dread or sadness.

The traditional approach to the rendering of clouds in video games uses static photographs or drawings of the real world sky. Usually, six photographs are used, each mapped on one side of a cube surrounding the game environment. This cube is called the skybox (Figure 1), it is rendered behind all other objects in the game and its center is always the player's position. If a sphere or hemisphere is used instead of a cube the object is called skydome. The static skybox can look good, but it has several problems. Weather and the time of day cannot be seamlessly changed, clouds cannot move based on the wind or the player position. There are several approaches to solve these issues.

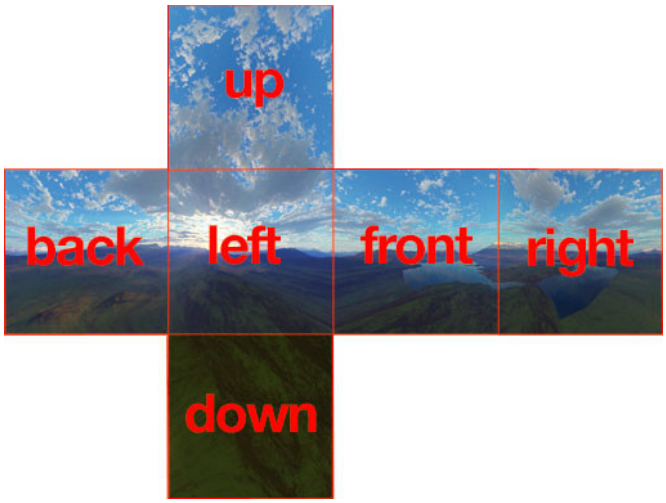


Figure 1: Example of a skybox texture

The game can have a library of cloud images. These can be then placed as textures on some geometry (typically quads). By moving the quads across the sky simulating the cloud movement. The flatness of the clouds, however, becomes visible, when the player or the clouds change their position. This can be solved by having multiple images for different view directions towards the cloud. A large amount of clouds is required for a realistic look and images start to take up a large amount of storage space. This method also has problems, when simulating changes in the cloud coverage. Clouds covering a larger amount of the sky can arrive from the distance, but clouds themselves will not emerge on the spot and their shapes will not change over time.

Another approach is generating clouds procedurally as shown by Roden and Parberry [13]. This allows for a dynamic change of coverage and shape of the clouds. Mukhina and Bezgodov [8] show, that realistic-looking clouds can be generated in 2D by simulating lighting effects such as light scattering and self-shadowing. The clouds however do not react to the changes of the player's vertical position.

Volumetric clouds can solve problems of the previously mentioned methods. The color of clouds can be obtained by using ray-tracing to follow the path of light that scatters multiple times in the volume. Novák et al. [11] describes this approach in more detail. Ray-tracing is used in offline rendering and it is too computationally expensive for real-time applications. Ebert et al. chapter 7 page 203 [3] uses a faster method – raymarching with single scattering. Single scattering, however, is only an estimation and does not produce completely physically correct results. Although faster, even this approach used to be too computationally heavy for real-time applications. Raymarching was slowed by a large number of texture reads. Recent advances in the consumer hardware performance, however, made the use of volumetric clouds possible.

The most important recent work on real-time volumetric cloud rendering is Andrew Schneider's implementation for the Decima game engine used by Guerrilla Games for their game Horizon Zero Dawn [14]. The shapes of the clouds were modeled from 3D noise textures and rendered by raymarching with single scattering. The raymarching was sped up by temporal reprojection and several other optimizations. Schneider was able to render the clouds in Full HD resolution under 2 ms.

Schneider's approach was improved upon by other authors. Hillaire [5] implemented volumetric clouds for the Frostbite engine used by EA. He improved lighting with a better multiple scattering approximation than single scattering. Schneider himself continued his work and the current cloud system is called Nubis [15]. It achieved several improvements, especially in lighting (more realistic dark edges and better multiple scattering approximation).

Apart from Schneider's method, there are other approaches to modeling and rendering of volumetric clouds. Yusov [21] generates clouds from smaller particles. Each particle has a specific position, density, radius, and color. Clouds used in the game Sea of Thieves [16] are just like any other object modeled from polygons. This gives an artist great control over the cloud shapes, but it is harder to simulate correct light transport in the clouds.

Goals

The aim of the thesis is to implement realistic real-time volumetric clouds similar to Schneider's work. The clouds are mapped as a flat texture on the skybox and are meant to be used in a game, where the player is positioned on the ground below them. This allows some optimizations but somewhat limits possible applications of the clouds. They shouldn't be used in flying simulators, or anywhere, where the player or other objects are positioned in the clouds. The work will be implemented as a package for the popular Unity game engine. Clouds will be tested on the graphics card GeForce GT 755M. The generated clouds should fulfill the following criteria:

- Automatically generated clouds unique in their shape and size
- Support physically-based lighting
- Three types of low altitude clouds – cumulus, stratus and stratocumulus

- Clouds change their shape over time
- Seamless transitions between different coverages and day times
- Clouds cast soft shadows and work with sun shafts

Thesis structure

This thesis is structured as follows. Chapter 1 covers the theory behind the volumetric cloud rendering. We take a look at different types of clouds and how to procedurally model them, then we discuss light transport in clouds and ways to simulate it on the computer. Chapter 2 describes the way cloud modeling and rendering are implemented in this thesis. Chapter 3 outlines various optimizations used in this Unity package to make the cloud rendering faster. The visual and performance results are shown in Chapter 4. the thesis concludes with discussion about the results and future work. Credit attribution for used images can be seen in the List of figures.

1. Theory

Methods used to generate procedural clouds for computer games do not simulate most of the physics of real-world clouds. It is, however, important to understand real-world cloud characteristics to model convincing virtual clouds.

1.1 Cloud Types

The information used in this section was taken from the Met Office [7] and the World Meteorological Organization[18].

Air always contains a certain amount of invisible water vapor. The amount of water vapor the air can hold depends on the atmosphere temperature. Warmer air can contain more of it. When there is more vapor, than the air can hold, it condenses to small water droplets (only about 10 micrometers in diameter). These droplets then attach themselves to small bits of dust and thus form clouds.

Apart from water droplets, clouds can also be formed by small ice crystals. Such clouds form in high altitudes, where the air is colder. Ice crystal clouds have a wispier appearance.

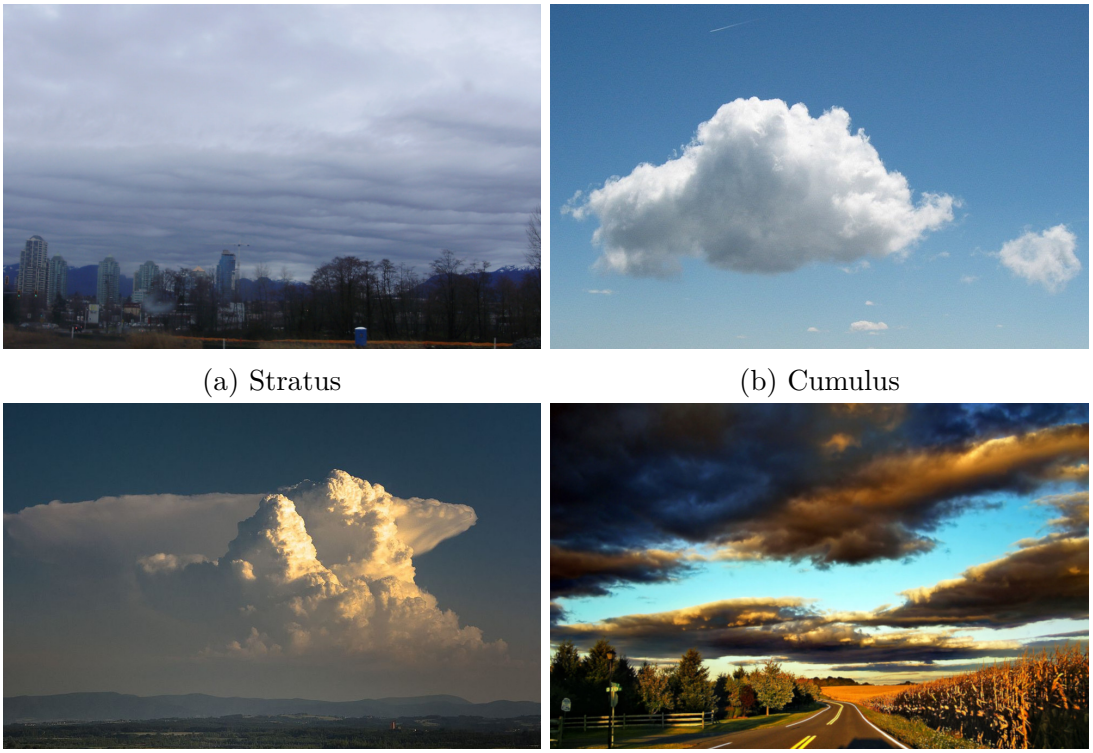
Not all clouds are the same, they differ in their shape, size, and altitude. The International Cloud Atlas [19] recognizes ten basic cloud types called "genera". Cloud genera are then be further subdivided into species (cumulus fractus, cumulus humilis...). Cloud genera can also be grouped according to the altitude of their base into three categories.

- Low-level clouds (Cumulus, Stratus, Stratocumulus, Cumulonimbus)
- Medium-level clouds (Altostratus, Nimbostratus, Altocumulus)
- High-level clouds (Cirrus, Cirrostratus, Cirrocumulus)

The cloud modeling in this thesis focuses on low-altitude clouds. Middle and high-altitude clouds appear flat from point of view of the player positioned on the ground. They can be therefore represented by 2D textures.

1.1.1 Low-level clouds

Low-level clouds have their base below 2000 meters. Stratus clouds (Figure 1.1a) form at very low altitudes (base below 400 meters). They tend to be uniform sheets with grey or white color, that can overcast the whole sky. Cumulus clouds (Figure 1.1b) are one of the most distinctive cloud types. These are formed by a hot air rising from the ground, which creates the characteristic cauliflower heads. If the cumulus clouds grow taller they can form cumulonimbus clouds (Figure 1.1c). The heads of these clouds can reach above 10km and take on a typical anvil shape. Stratocumulus clouds (Figure 1.1d) usually form by breaking of the stratus cloud layer, but they can also form by a cumulus cloud spreading out. They tend to be smaller and thinner than cumulus clouds and do not have the characteristic cauliflower head.



(a) Stratus (b) Cumulus

(c) Cumulonimbus (d) Stratocumulus

Figure 1.1: Low altitude clouds

1.1.2 Medium-level clouds

Medium-level clouds have their base level between 2000 and 6000 meters. Altostratus clouds (Figure 1.2a) have an appearance of uniform color sheets. Their look is similar to lower altitude stratus clouds, but they do not completely block the Sun and its disk can be seen behind the Altostratus layer. Nimbostratus clouds (Figure 1.2b) are dark thick, featureless layers of clouds producing continuous raining or snowing. Although they are formed in the medium cloud layer, they often extend both into the low layer. Altocumulus clouds (Figure 1.2c) are patches of usually round cloudlets.

(a) Altostratus (b) Nimbostratus (c) Altocumulus

Figure 1.2: Medium altitude clouds

High-level clouds

High-level clouds have their base above 6000 meters. Cirrocumulus clouds (Figure 1.3a) is a rare type of clouds. They are visually similar to Altocumulus clouds but even smaller. Cirrostratus clouds (Figure 1.3b) look like a white very thin semitransparent veil. They are formed from ice crystals. Cirrus clouds (Figure 1.3c) resemble wispy strands. These clouds are made up of ice crystals.

(a) Cirrocumulus (b) Cirrostratus (c) Cirrus

Figure 1.3: High altitude clouds

1.2 Clouds representation and modeling

There are many possible representations of clouds. Just like the rest of the geometry, the cloud shapes can be described by a polygon mesh. The polygonal representation was implemented in the game Sea of Thieves [16]. This approach gives an artist control over the shapes of clouds. There are, however, some problems. It does not store density at each point in the cloud, just its boundaries. Advanced lighting of clouds is therefore hard to simulate.

Clouds can also be represented by particles, each represented by position, radius, density, and color as shown by Yusov [21].

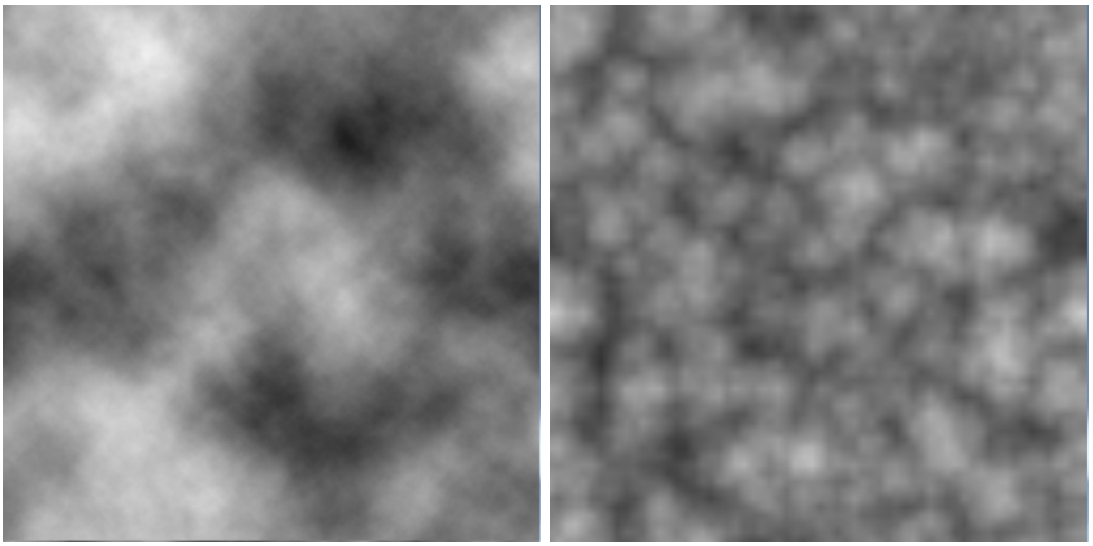
A straightforward approach is to store cloud density values into a 3D texture. If, however, used for detailed clouds it quickly becomes too memory heavy.

Clouds can be described by a procedurally generated noise function. The noise has to be, however, computed at every sampled point, which makes its use too slow.

Usually, a combination of these representations is used. Kniss et al. [6] and Ebert et al. [3] model clouds using implicit volumes and a perturbation texture. The idea is to model the basic cloud shapes using implicit functions describing geometrical primitives (spheres, ellipsoid, cubes...). To add detail to the clouds a procedurally generated 3D noise is used to alter the cloud density. The noise can be stored in a 3D texture to speed up the sampling.

Schneider [14] uses a series of different 2D and 3D textures, which are combined to form the cloud shapes. A 2D cloud map is used to roughly specify the areas, where the clouds are going to appear. The main shape is defined by a combination of inverted layered Worley [20] noise (Figure 1.4b) and a layered

Perlin [12] noise (Figure 1.4a). Details are added by another 3D layered Worley noise texture.



(a) Layered Perlin noise (b) Layered Worley noise

Figure 1.4: Noises used for cloud modeling

1.3 Light transport in clouds

When light enters a cloud it interacts with the water droplets and ice crystals in the cloud in two ways: it can either be absorbed or scattered by these particles. In light transport theory, particles interacting with light are called the participating medium.

1.3.1 Absorption

Light can be absorbed into the medium along the way. The amount of absorbed light can be determined by the Beer-Lambert law.

$$A = \epsilon c l \tag{1.1}$$

Where absorbance A linearly depends on the molar concentration c and the absorption ability ϵ of the particles and the distance the light travels through the medium l. Absorbance isn't very useful to describe what part of light was absorbed, transmittance T should be used for this purpose. Equation 1.3 calculates transmittance in a homogeneous medium.

$$A = -log_{10}T \tag{1.2}$$

$$T = e^{-\beta_a l} \tag{1.3}$$

Where β_a is an absorbtion coefficient combining both ϵ and c from the original Beer-Lambert law.

This means, that the amount of light transmitted through the cloud decreases exponentially with distance. Homogeneous medium isn't a good aproximation of real-world clouds. Gas density in a nonhomogeneous medium like clouds varies and different equation is necessary to describe transmittance. Equation 1.4 calculates it between the points s_1 and s_2, where $\beta_a(s)$ is an absorbtion coefficient at the point s.

$$T = e^{-\int_{s1}^{s2} \beta_a(s)ds} \tag{1.4}$$

Because of the Beer-Lambert law, the parts of clouds further from the Sun appear darker, this is called self-shadowing. Although the amount of transmitted light decreases exponentially, human eyes perceive the decrease as approximately linear [17], this can be seen in Figure 1.5.

Figure 1.5: Self shadowing in clouds

1.3.2 Scattering

When light enters the cloud it can be scattered (a process where light is forced to deviate from the straight trajectory) by water droplets and ice crystals. Light scattering happens multiple times and the probability of scattering is similar for all visible wavelengths – this is the reason for the white color of clouds [7].

Scattering in the clouds is anisotropic, cloud particles tend to scatter the light forward (in a direction similar to the original). As a result, clouds closer to the Sun dot have silver linings and are generally lighter (see Figure 1.6a). The probability of the light from direction ω_2 to scatter in the direction ω depends on the angle between the two vectors and can be described by a phase function $f_p(\omega, \omega_2)$. The phase function is also dependent on the wavelength of incoming light, but in the case of visible light interaction with cloud particles, they are very similar.

Many different functions can be utilized depending on characteristics of the participating medium. For larger particles, the clouds are composed of, using the Mie scattering theory produces the most accurate phase functions. They are usually difficult to work with because of their complexity. Simpler functions give approximations sufficient for real time applications. Hadwiger et al. [4] suggests using renormalized Henyey-Greenstein phase function:

$$f_p(\omega, \omega_2) = \frac{1 - g^2}{(1 + g^2 - 2g\frac{\omega \cdot \omega_2}{|\omega| \cdot |\omega_2|})^{3/2}} \tag{1.5}$$

Where g is a parameter describing, whether the forward or backward scattering is dominant. A positive value of g means, that most of the light scatters forward and should be therefore used for light transport in clouds. To estimate cloud lighting, two types of scattering are important.

Out-scattering – light scatters away from the eye ray. The amount of out-scattered light is dependent on the density and composition of the cloud at the point. The computation is the same as in the case of absorption (1.6).

$$T = e^{-\int_{s1}^{s2} \beta_s(s) ds} \tag{1.6}$$

Thanks to the linearity of absorption and out-scattering, their coefficiens are be combined into a single extinction coefficient in Equation (1.7).

$$\beta_e = \beta_a + \beta_s \tag{1.7}$$

$$T = e^{-\int_{s1}^{s2} \beta_e(s) ds} \tag{1.8}$$

In-scattering – light scatters in the direction of the eye ray. This light can come directly from the Sun, or it might be the light scattered in the cloud. Points inside the cloud have more scattered light than the ones near the surface. This is the reason, the edges of the clouds appear darker than the rest of the cloud (see Figure 1.6b).

(a) Silver lining near the Sun dot (b) Dark edges of the clouds

Figure 1.6: Cloud lighting effects

1.4 Cloud Rendering

Volumetric data can not be rendered the same way solid objects are. To get the right cloud color, the program must solve the Volume Rendering Equation 1.9 (as shown by Novák et al. [11]).

$$L(x,\omega) = \int_0^z T(x,y)[\sigma_a(y)L_e(y,\omega) + \sigma_s(y)L_s(y,\omega)]dy + T(x,z)L_o(z,\omega) \quad (1.9)$$

Where $T(x,y)$ describes the transmittance between the point x and y (the fraction of light from the point y, that reaches the point x). The transmittance can be calculated using the following equation.

$$T(x,y) = e^{-\int_0^y \beta_e(s)ds} \quad (1.10)$$

$\sigma_a(y)L_e(y,\omega)$ describes the light emission by the volume. This can be used when rendering explosions, but clouds do not emit light.

$$\sigma_a(y)L_e(y,\omega) = 0 \quad (1.11)$$

$\sigma_s(y)L_s(y,\omega)$ describes the in-scattering (light scattering towards the eye). The amount of in-scattered light can be computed using the following formula.

$$L_s(y,\omega) = \int_{S^2} f_p(\omega,\omega_2)L(y,\omega_2)d\omega_2 \quad (1.12)$$

Where $f_p(\omega,\omega_2)$ is a phase function describing the probability of the light from direction ω_2 to scatter in the direction ω. Henyey-Greenstein phase function (1.5) can be used for this purpose. $L_o(z,\omega)$ describes the amount of light directed in the eye direction by the object behind the cloud volume.

The volume rendering equation is recursive, because $L(y,\omega)$ is computed by another volume rendering equation. This corresponds to the light scattering multiple times inside the cloud. The equation can be numerically solved using the ray-tracing [1]. Algorithm corresponds to program following the light scattering multiple times in the cloud. This approach is often called multiple scattering and gives very convincing results. Multiple scattering is however too slow for real-time use.

To render convincing clouds, it is not necessary to correctly solve the volume rendering equation. An estimation similar to the real result is usually sufficient. This estimation can be obtained by using ray marching and single scattering.

Ray marching is a popular method of rendering volumetric data [9]. Samples of the volumetric media are taken at regular intervals (see Figure 1.8). In a mathematical sense, this method corresponds to computing an integral using deterministic quadrature.

Figure 1.7: Multiple scattering clouds by Bouthours et al. [1]

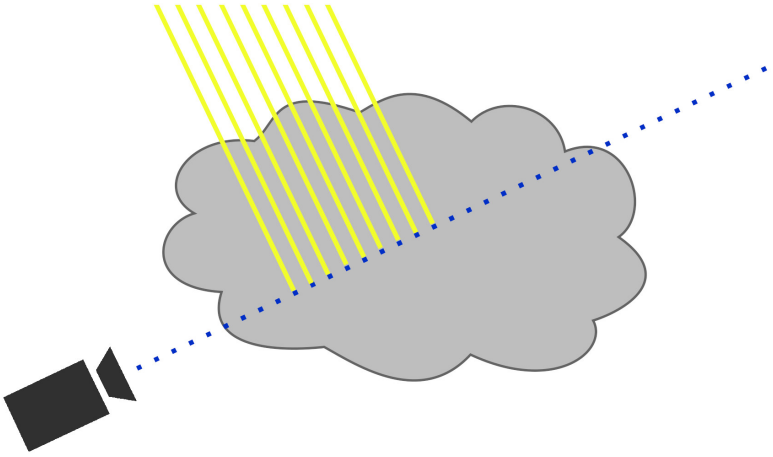


Figure 1.8: Visualization of ray marching with single scattering (samples - blue dots, light coming to the sampled point directly from the Sun - yellow lines)

Sampled point receives lighting arriving in a polyline, with scattering happening at each of its points. Single scattering is an estimation, that takes in only the lighting coming directly from the Sun. In order to get results similar to those obtained by multiple scattering, the extinction coefficient must be lowered. In real clouds, light arrives at the sampled point both directly from the Sun and indirectly from other parts of the clouds. Lowering the extinction coefficient artificially adds extra light compensating for ignored indirect lighting. The extinction coefficient should be lowered to a value slightly higher than the absorption coefficient.

2. Implementation

JK Volumetric Clouds is a package developed for the Unity game engine. It allows user to add clouds to the standard procedural skybox. Different low-level clouds are implemented, the sky coverage can be easily modified, the clouds move and evolve over time and cast soft shadows and light shafts. The package contains shaders written in HLSL and scripts in C#. The shaders are used to render the clouds and sunshafts, while the scripts control parameters passed to the shaders and the rendering order.

2.1 Cloud modeling

The program models three types of low altitude clouds – cumulus, stratus, and stratocumulus. Middle and high altitude clouds are not implemented, because they appear flat from point of view of the player positioned on the ground. Simple 2D textures can be used for them to save computation time. Cumulonimbus isn't implemented, because it reaches up to 8km in height, which would greatly increase the number of necessary ray marching steps.

The clouds are modeled by specifying their density at each point in space. The density is determined from different 2D and 3D textures and mathematical functions. Simplified process of the cloud density calculation can be seen in 2.1 and consists of the following steps:

1. A 2D texture called cloud map determines the type and position of the clouds. It only gives a rough outline of the cloud shape.
2. Shape altering function is used to describe the vertical position of different cloud types and also makes the clouds rounded at the top and bottom.
3. 3D shape noise is added to determine the shape of the cloud.
4. 3D detail noise is used to add small details to the edges of the clouds.

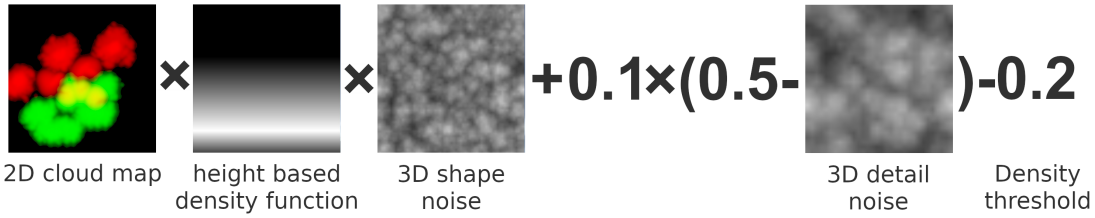


Figure 2.1: Computation of the cloud density

2.1.1 Cloud map

The cloud map is a 2D texture, that controls the position, type, amount, and density of the clouds. Only a horizontal position is used to sample the cloud map. It contains information in all 4 of its channels. Each of the 3 color channels corresponds to one of the cloud types – red to cumulus, green to stratus, and blue

to cumulostratus. The value stored in the channel gives the maximum possible cloud density (before applying the threshold) at a specific horizontal position. The cutoff threshold is set to 0.2, so the areas with a lower cloud map value in all channels have no chance of clouds appearing, the value of 1 on the other hand almost certainly means, that the cloud will appear. The alpha channel stores cloud coverage. Cloud coverage doesn't in any way impact the presence of clouds, it is a variable used when generating the cloud map procedurally. In the areas with higher coverage, the program places higher values into the color channels. Two types of cloud maps can be distinguished.

User-supplied

The cloud map is a texture given to the program by the user. A static texture gives clouds a fixed position and size. Cloud shape is still evolving, but the clouds are not be moved by the wind, and coverage does not change. Animated texture can be supplied by the user to simulate these effects. Example of a user created cloud map can be seen in Figure 2.2a.

Generated

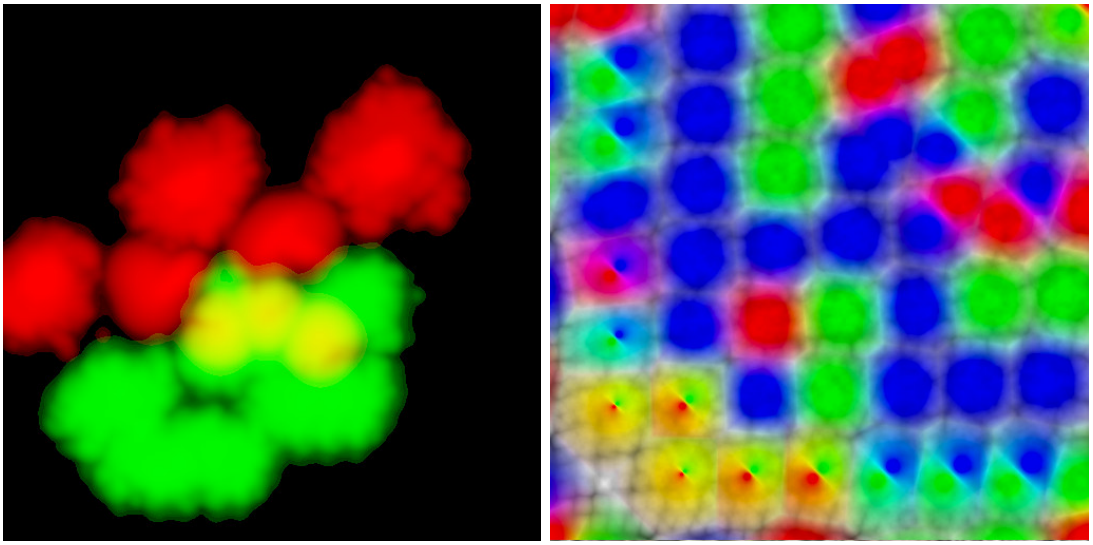
A generated cloud map is a texture created by the program itself. It is generated using a combination of two noise function – layered Perlin noise and layered Worley noise. The Perlin noise defines wispy shapes, while the Worley noise defines separate clouds and adds billow shapes to them. Separate clouds are defined by the least detailed layer of the Worley noise and each of the clouds has a randomly assigned type. User can specify the probability of different clouds to appear. An example of an automatically generated cloud map can be seen in Figure 2.2. The resolution used in this implementation is 512x512. Sharp details aren't necessary because the cloud map is only used to roughly define the basic shape of clouds and a higher resolution would significantly slow down the generation process.

An initial sky coverage can be specified by the user. Higher coverage increases the radius of the Worley noise cells generating bigger clouds, that cover a larger part of the sky. The cloud position moves based on the set wind speed and direction and their shape evolves. The sky coverage can be changed at run-time using the *Coverage Change To* parameter. The coverage of the clouds newly appearing on the horizon will slowly move closer to the desired value.

2.1.2 Height dependent shape altering functions

The program renders all clouds between 500 and 2000 meters above the ground, but most clouds do not fill all of this space. Stratus clouds form a thin layer near the bottom of this interval and stratocumulus clouds fill about half of this interval. Clouds also need to be rounded towards the top and bottom. To model this a height dependent functions are used.

Each function takes in the vertical position of the sample in the clouds layer, where 0 corresponds to 500 meters and 1 to 2000 meters. The function then returns the probability of clouds appearing at particular height. Height dependent functions for different cloud types can be seen in Figure 2.3.



(a) Static user supplied cloud map (b) Automatically generated cloud map

Figure 2.2: Examples of cloud maps

(a) Cumulus (b) Stratocumulus (c) Stratus

Figure 2.3: Shape altering height functions for different cloud types

Together with the cloud map, these functions give the basic shape of the clouds. Visualization of the cloud map combined with the height function for the cumulus clouds can be seen in Figure 2.4.

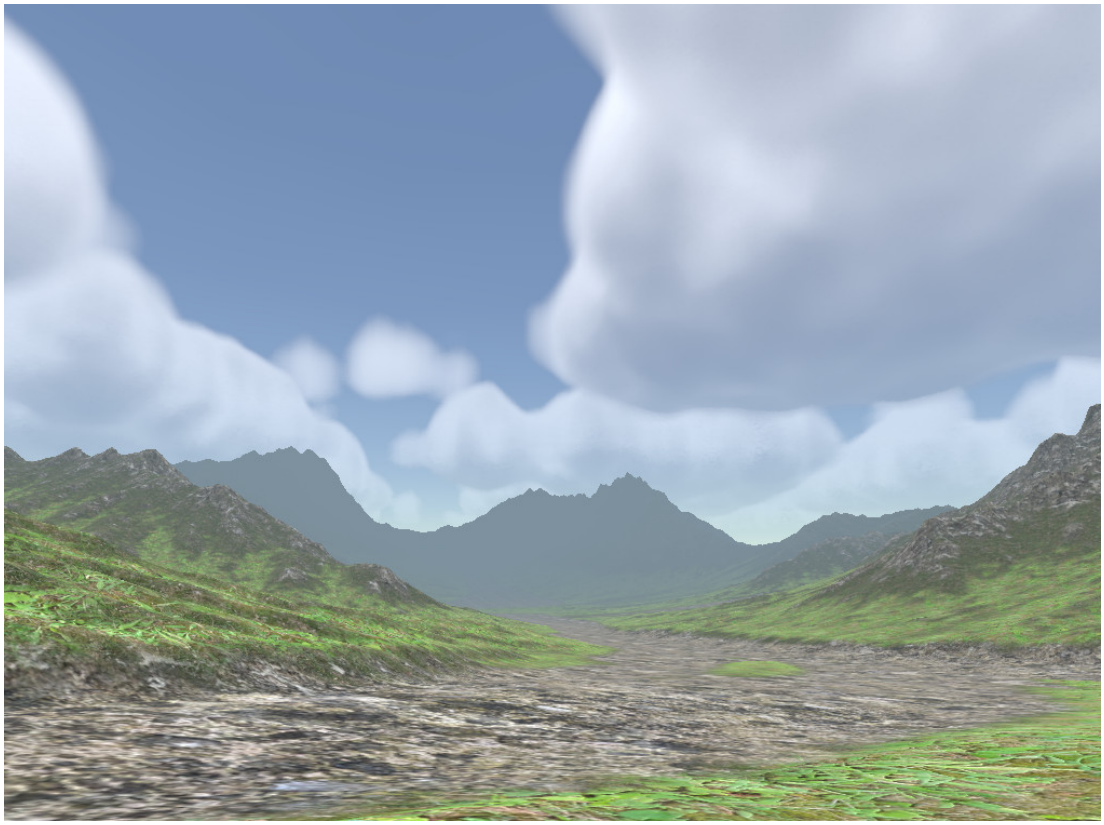
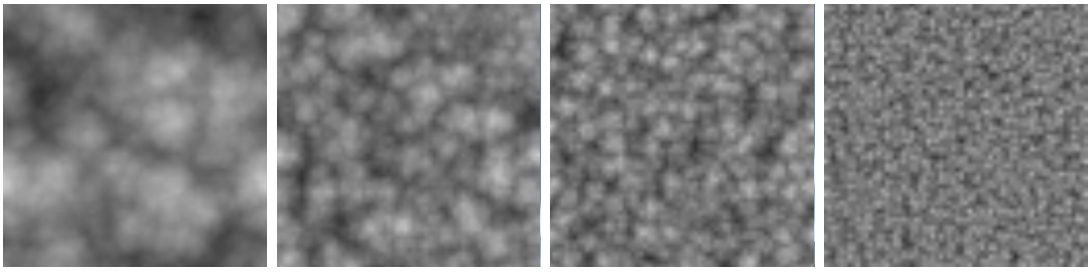


Figure 2.4: Shape of the clouds defined by the cloud map and the shape altering height function

2.1.3 Shape noise

The cloud map and the height function gives a rough shape of the cloud, but do not offer enough variation. Cloud needs a more defined shape. This shape is carved out using the shape noise.

Shape noise is a layered Worley noise stored in a 64x64x64 texture. Different noise frequencies are stored in four color channels. Noises stored in these channels are shown in Figure 2.5.



(a) Red channel (b) Green channel (c) Blue channel (d) Alpha channel

Figure 2.5: Color channels of the shape noise texture

The noises in all of the 3D texture channels are combined to add both bigger and smaller shapes to the clouds. Different weights given to channel samples are an artistic decision.

The resulting shape noise sample is computed using equation (2.1) where, s_r is the noise from the red color channel, s_g noise from the green channel, s_b noise from the blue channel, and s_a from the alpha channel.

$$SN_{sample} = 0.25s_r + 0.25s_g + 0.45s_b + 0.05s_a \tag{2.1}$$

The shape noise sample is then combined with the cloud map and the height dependent shape altering function in (2.2) (0.2 is treshold for the clouds to appear picked as an artistic decision).

$$Sample = SN_{sample} \cdot CloudMap \cdot HeightFunction - 0.2 \tag{2.2}$$

The result is then further changed depending on the cloud type. The resulting sample for cumulus and stratocumulus clouds is multiplied by 2 to give these clouds more defined shapes. The sample for stratus clouds is further multiplied by the cloud map sample to add more variations to the cloud density. The result of applying the shape noise on the cumulus clouds can be seen in Figure 2.6.

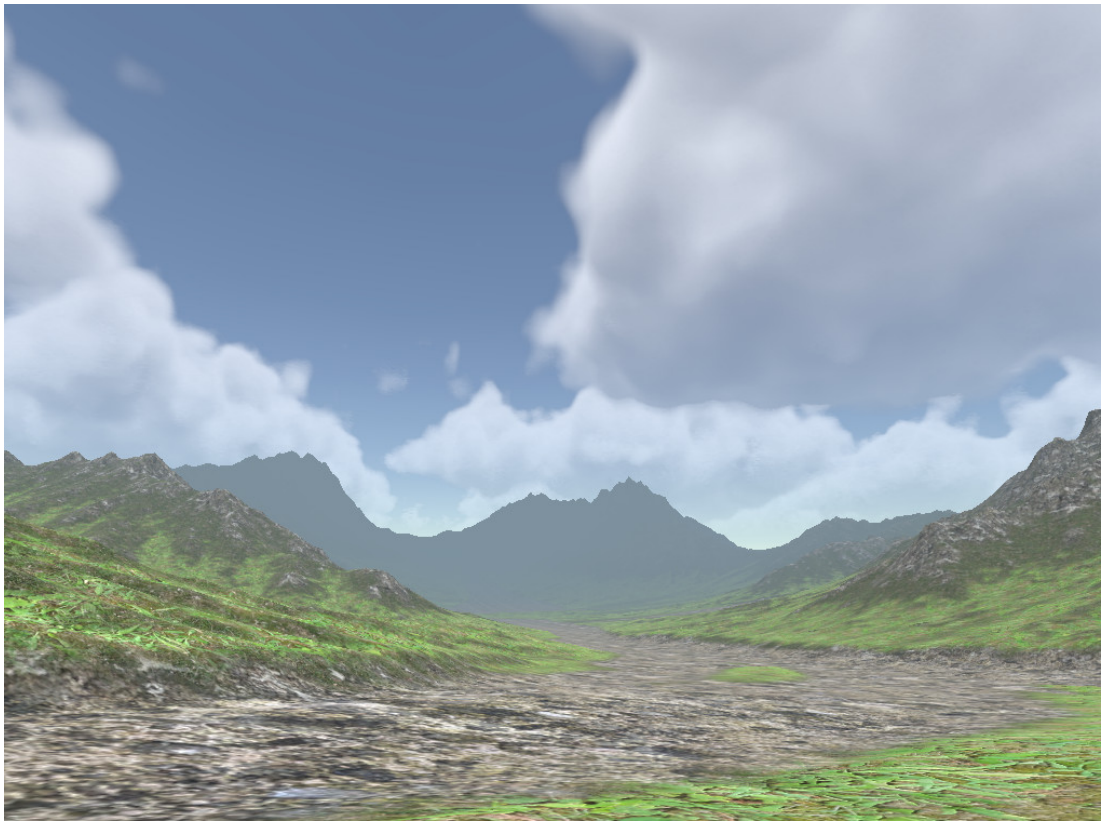


Figure 2.6: Clouds after applying shape noise

2.1.4 Detail noise

The shape of the clouds is now more varied, but the edges are still too blurry, while in real clouds they appear whispy. Sharper edges are added using a 3D detail noise texture at resolution 32x32x32 similar to the shape noise.

The detail noise is subtracted from the original sample, this adds whispy shapes instead of the puffy ones created from the shape noise. The noise application strength varies with distance. The detail texture is very visible close to

the observer, while it is almost completely removed in distance. The detail there in there is so small, that it doesn't add variation to the edges of the clouds, but only noise. The result can be seen in Figure 2.7

Figure 2.7: Clouds after applying detail noise

2.1.5 Cloud movement

Clouds are moved across the sky based on the speed and direction of the wind. These variables are specified by the user. The movement is simulated by offsetting the cloud map (only when generated), shape noise, and detail noise. Each texture is moved at a different speed, which makes the cloud shapes evolve. The procedural cloud map automatically generates shapes of the newly appearing clouds.

The user-supplied cloud map is not offset by the wind, because the way to generate newly appearing clouds isn't clear. Clouds generated using such maps do not move over the sky, but they still change shape based on the movement of the 3D noise textures. The user can supply an animated texture, simulating more complicated cloud movements.

2.2 Visualization and lighting

The clouds are rendered as a projection on the Unity skybox. This makes it easier for the user to implement things like reflections in the water, but traveling through the clouds, or placing objects in them is impossible. For example, when there is a mountain with its top higher than the cloud layer, the clouds are still

rendered above it. If the player would come on top of such a mountain, all parts of the clouds below the horizon would get discarded. Because the program is meant for games, where the player is positioned on the ground it only expects the clouds to appear in the upper hemisphere.

The clouds are visualized using ray-marching with single scattering lighting. The clouds support lighting effects, such as casting shadows and light shafts.

2.2.1 Ray marching

To render physically accurate volumetric clouds the program must solve the volume rendering equation 2.3 (Novák et. al [11]).

$$L(x,\omega) = \int_0^z T(x,y)[\sigma_a(y)L_e(y,\omega) + \sigma_s(y)L_s(y,\omega)]dy + T(x,z)L_o(z,\omega) \quad (2.3)$$

Offline rendering uses methods like ray tracing to solve this equation. These methods are however visually unstable and their computation too slow. The priority for real-time rendering is speed and simplicity. The result just needs to be a relatively close estimate of the equation solution. One of the methods used to estimate integrals is deterministic quadrature (2.4). It works well on integrals where the value of the integrated function does not change too much on small intervals.

$$L(x,\omega) = \sum_{i=1}^{n} T(x,y)[\sigma_a(y_i)L_e(y_i,\omega) + \sigma_s(y_i)L_s(y,\omega_i)]\Delta y + T(x,z)L_o(z,\omega) \quad (2.4)$$

This corresponds to the ray marching rendering method. Steps are taken at a fixed interval in the direction of the eye ray. The current implementation of the program takes a fixed number of 64 steps taking a cloud density sample at each of them. The value of the integrated function is computed at each sample. First thing to compute is the transmittance (2.5). The transmittance decreases exponentially according to the Beer-Lambert law.

$$T(x,y) = e^{-\int_0^y \beta_e(s)ds} \quad (2.5)$$

The raymarching algorithm simplifies this function by assuming, that the cloud density is uniform between the sampled points. The transmittance equation can then be transformed into (2.6).

$$T(x,y) = e^{-\sum_{i=1}^{n} \beta_e(y_i)\Delta y} \quad (2.6)$$

Moreover the transmittance value from the previous sample T_{m-1} can be reused to compute the current transmittance. This greatly simplifies the equation (2.7).

$$T_m = T_{m-1}e^{-\beta_e(y_m)\Delta y} \quad (2.7)$$

Light emission in the clouds is always equal to zero, so the only thing left to compute is the in-scattering. The estimation of its value will be discussed in the lighting section.

When raymarching starts at the same height for each point in the sky, a banding pattern appears (Figure 2.8a). The problem can be solved by offsetting the start of each raymarching by a pseudorandom distance (Figure 2.8b).

(a) No offset (b) With offset

Figure 2.8: Clouds with and without a random offset at the start

2.2.2 Lighting

In scattering needs to be calculated at each point sampled by raymarching. To correctly calculate the lighting at that point, the in-scattering equation (2.8) must be solved.

$$L_s(y,\omega) = \int_{S^2} f_p(\omega,\omega_2)L(y,\omega_2)d\omega_2 \tag{2.8}$$

It is necessary to compute this integral over the whole sphere, moreover $L(y,\omega_2)$ leads to a volume rendering equation (2.9), making the in-scattering equation recursive.

$$L(y,\omega_2) = \int_0^{z_2} T(y,y_2)[\sigma_a(y_2)L_e(y_2,\omega_2) + \sigma_s(y_2)L_s(y_2,\omega_2)]dy_2 + T(y,z_2)L_o(z_2,\omega_2) \tag{2.9}$$

This is too complex for a real-time application, some drastic approximation is needed. Such approximation is the single scattering. It is assumed, that all of the in-scattered light comes directly from the Sun direction. The equation is then approximated as (2.10).

$$L_s(y,\omega) = f_p(\omega,\omega_{Sun})T(y,Sun_{pos})L_{Sun}(Sun_{pos},\omega_{Sun}) \tag{2.10}$$

Beer-Lambert law

Transmittance in the in-scattering equation can be computed using the Beer-Lambert law (2.11).

$$T(x,y) = e^{-\int_0^y \beta_e(s)ds} \tag{2.11}$$

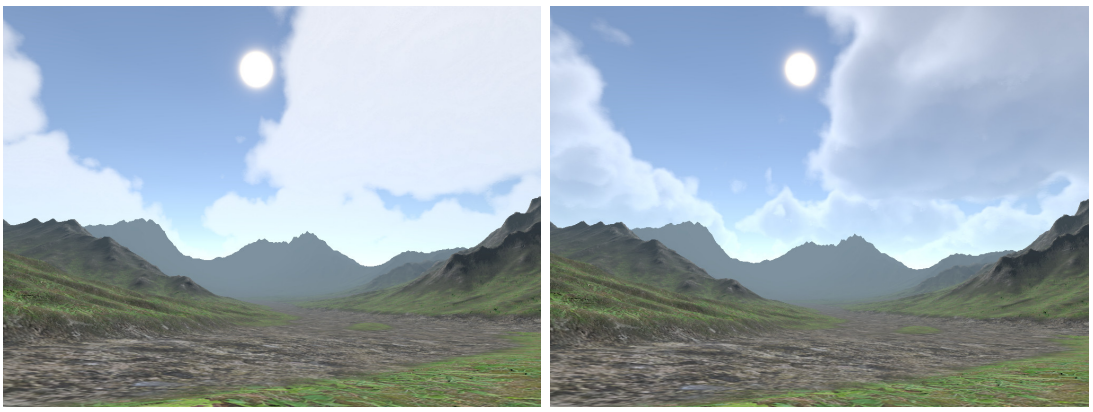
To make up for the in-scattered light, that does not come directly from the Sun, the extinction coefficient is lowered compared to the one used in the volume rendering equation. This isn't physically correct but moves the clouds visually closer to their real-world counterparts. The amount the extinction coefficient is an artistic decision. In this thesis, the coefficient used for stratus remains unchanged, while for cumulus and cumulonimbus it is set 0.4 of the original value.

The integral can be approximated using raymarching, which transforms the integral into a sum (2.12). These raymarching steps will be called the self-shadowing steps.

$$T(x, y) = e^{-\sum_{i=1}^{n} \beta_e(y_i)\Delta y} \tag{2.12}$$

The number of self-shadowing steps required to get satisfying results is much lower than the necessary number of cloud marching steps. The implementation used in this thesis takes 6 steps towards the Sun exponentially increasing in length.

Calculating the light transmitted to the sampled point from the Sun corresponds to parts of the cloud throwing shadows on the sampled point. The visual result of applying the Beer-Lambert law to the light coming from the Sun can be seen in Figure 2.9.



(a) without Beer-Lamber law (b) with Beer-Lamber law

Figure 2.9: Cloud self shadowing after applying the Beer-Lambert law

Heyney-Greenstein phase function

There a problem with these clouds. They do not have silver linings near the Sun dot like the real clouds. This is when the phase function ($f_p(\omega, \omega_{Sun})$) part of the in-scattering equation comes in. Phase function used in this implementation is the Heyney-Greenstein function (2.13), as suggested by Hadwiger et al. [4], with parameter $g = 0.7$. A positive value of g means, that the light tends to scatter forward.

$$f_p(\omega, \omega_2) = \frac{1 - g^2}{(1 + g^2 - 2g\frac{\omega \cdot \omega_2}{|\omega| \cdot |\omega_2|})} \tag{2.13}$$

After applying the Heyney-Greenstein phase function silver linings around the Sun dot appear (see Figure 2.10).

Shadows

Clouds cast semitransparent shadows. They are rendered using a quad called shadow plane. This quad is completely transparent and the player does not interact with it, but some of its fragments specified by the SHADOW_CASTER_FRAGMENT() function cast shadows. The problem is, that these shadows always have the full strength. Shadow strength should vary based on cloud thickness and density.

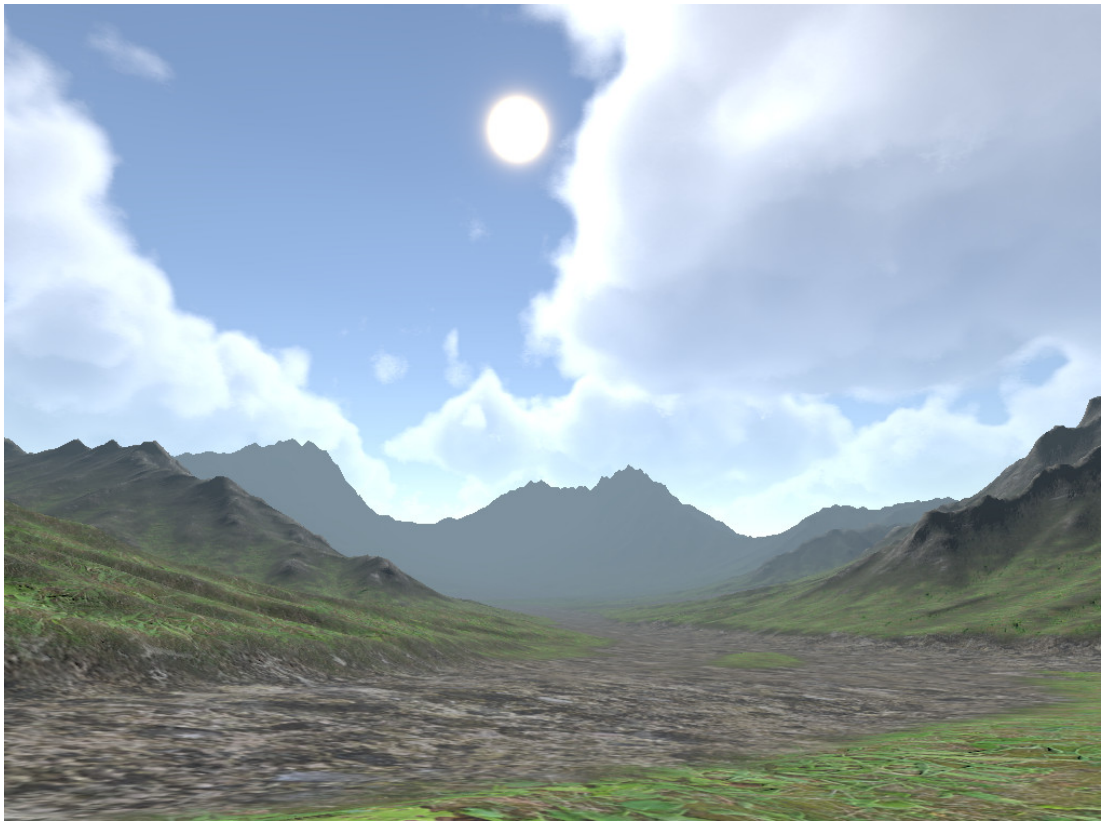


Figure 2.10: Clouds after applying the Heyney-Greenstein phase function

Semitransparent shadows are achieved using dithering. The dithering pattern becomes almost unnoticeable, when the soft shadows are used.

The strength of the shadows is determined using the cloud map, which is quite similar to the real cloud shapes. The Figure 2.11 shows different sky coverages casting shadows.

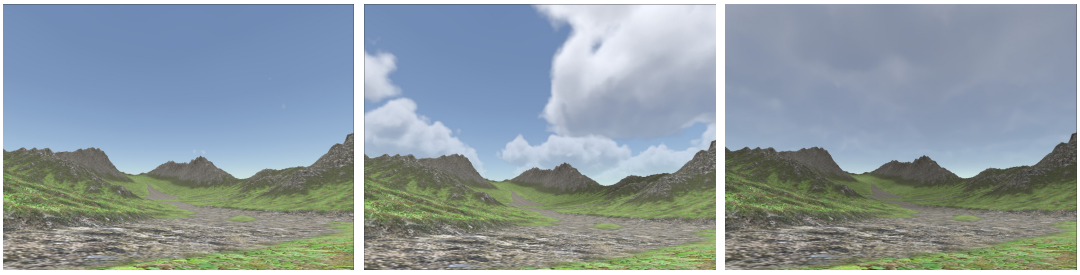


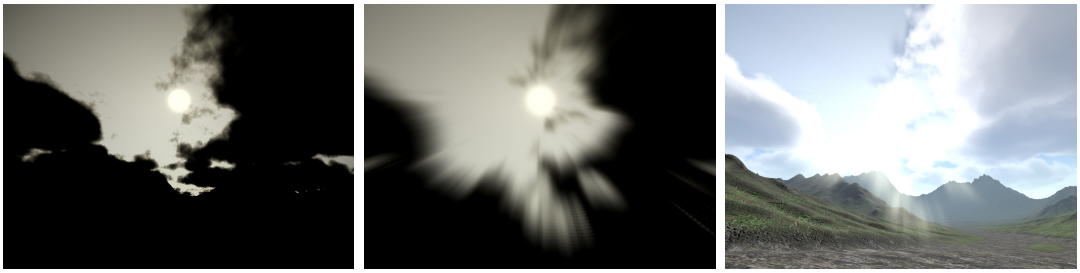
Figure 2.11: Shadows cast by different sky coverages

Lightshafts

Sunshafts are implemented as a post-processing effect described by Nguyen [10]. These sun shafts do not exactly correspond to their real-world counterpart, but they are faster to render than truly volumetric sunshafts.

The shaft rendering consists of three steps: first, the program selects potential sources of light shafts. These are areas with a clear sky unobstructed by any objects. The color of these areas is modified to match the intended light shafts

color (Figure 2.12a). These are then blurred in the direction away from the Sun dot. This creates an effect of sun shafts peaking over objects, or through the clouds (Figure 2.12b). Finally, the blurred image is then added to the originally rendered image. Figure 2.12c shows extremely strong light shafts (to make the effect more visible) over the mountain scene.



(a) clear parts of the sky selected as light sources

(b) Light sources blurred away from the Sun

(c) Lightshafts added to the original image

Figure 2.12: Sunshafts generation process

Ambient light

Ambient is a non physical light coming uniformly from all the directions used to describe lighting arriving from different sources, than the Sun. It can be for example used to light up the clouds on the night sky (describing light coming from the stars). The implementation used in this thesis linearly interpolates between two ambient light colors based on the altitude of the sampled point. The upper light represents light coming from the skybox (stars in the night, blue sky in the day...). The lower ambient represents light coming from the ground. Both ambient light colors are selected by the user.

2.2.3 Aerial perspective

To create a realistic scenery atmosphere must be taken into account. Blending clouds with the atmosphere gives them a more realistic look and a sense of depth. A realistic atmosphere can be generated by computing light scattering. Bruneton [2] uses precomputed atmospheric scattering to a great effect. The approach used in this thesis is a simple blending of clouds with the color of the underlying skybox. The amount of the skybox color used depends on the distance to the clouds. More skybox color is used further to the horizon until only the skybox color is used. Results of the atmosphere blending can be seen in Figure 2.13.

(a) Without atmosphere blending (b) With atmosphere blending

Figure 2.13: Comparison of the clouds with and without the atmosphere blending

3. Optimization

The Cloud visualization described in the previous chapter does not run in real-time. Some drastic optimizations are needed to get acceptable render times.

The slowest part of the cloud rendering is the high number of texture samples. For each rendered pixel there are 64 raymarching steps. Each of these steps samples two 3D and one 2D texture. It also computes lighting, sampling these textures again in 6 self-shadowing steps.

There are two main ways to increase rendering speed. Decrease the number of pixels rendered per frame and decrease the number of texture reads during the raymarching.

3.1 Raymarching optimization

3.1.1 Cloud density sampling

The density of the clouds at a certain point is determined by three textures and one function. Not all of them however need to be sampled all the time. It can be clear, that there are no clouds at the sampled position just based on the cloud map and the height function.

Rendering of the sky in Figure 3.1 can be sped up. Stratus clouds are located only in a thin layer, so samples at different heights can be entirely skipped. The sky is also covered by the clouds only sparsely and some of the areas without clouds can be identified just by using the cloud map.

Shape and detail noise sampling can be skipped in both of these cases. This makes the shader significantly faster because two out of the three texture samplings are skipped.

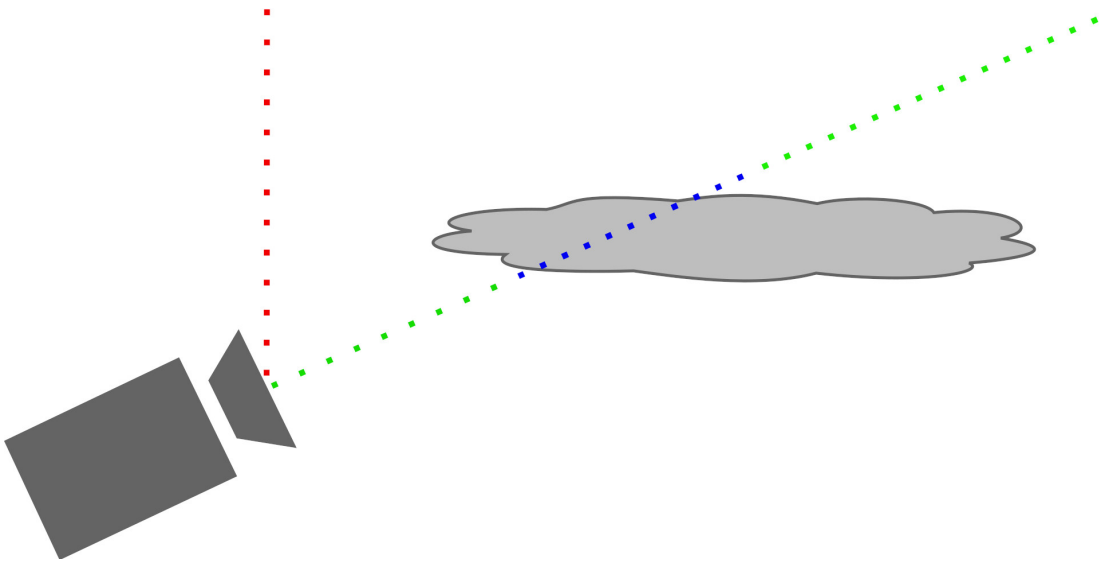


Figure 3.1: Sky optimized by skipping of the 3D texture sampling. There are clearly no clouds at green (because of height function) and red (based on the cloud map) sample points.

The detail noise texture is used to make the edges of the clouds sharper and more detailed. There is no need to use it on the clear sky, where there is no chance for a cloud to appear or in the middle of a cloud.

The cloud density sampled before applying detail noise can have both positive and negative values. If the value at the sampled point is clearly positive, the cloud will appear there regardless and detail noise will only slightly adjust its density. A clearly negative value on the other hand gives no chance for a cloud to appear at all. The detail noise should be only sampled, when the density is close to zero. Detail noise can then be important in deciding if the sampled point contains clouds. This way, the detail noise is not used in most samples.

3.1.2 Self-shadowing sampling and lighting

Lighting only needs to be calculated when the cloud density at the sampled point is non-zero.

Sampling clouds for self-shadowing has some extra optimization. Detail noise sampling can be completely skipped, because small wisps at the edge of the clouds have little impact on the cloud self-shadowing.

3.1.3 Early exit

As the ray marcher goes deeper into the cloud, less light arrives from the sampled point to the observer's eye. These points contribute very little to the final color of the cloud and don't have to be rendered. In the volume rendering function it is represented by decrease in transmittance.

If the transmittance is sufficiently low, the marching can be stopped early. This optimization is especially useful for overcast skies, where almost all of the raymarching stops early (Figure 3.2a).

3.1.4 Longer steps

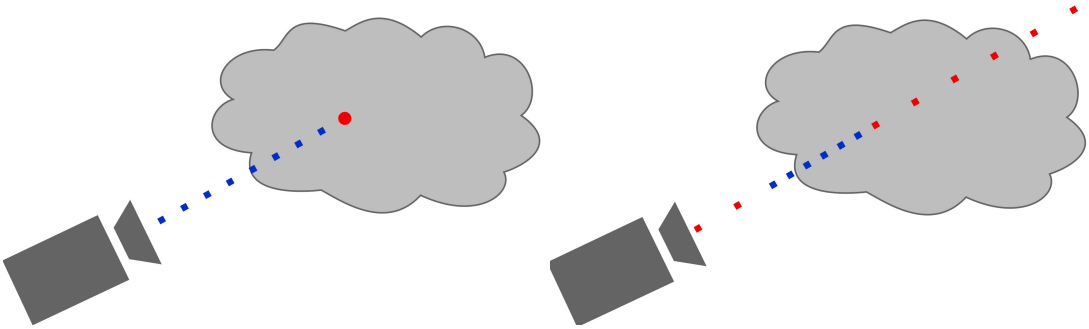
Raymarching steps need to be short enough to be able to render detailed parts of the cloud. However in some parts of the ray, longer steps would be sufficient.

Two lengths of steps are used, the normal unoptimized step, and a two times longer step. The longer steps are taken, when the raymarching is outside of the cloud, or deep inside the cloud, where details become unimportant (Figure 3.2b).

The raymarching starts with the longer step length and continues using it until it reaches a cloud. When the cloud is reached the sampling takes a smaller step back (in case part of the cloud was missed) and switches to smaller steps. The program switches back to longer steps if the sampling exits the cloud, or the transmittance falls under 0.1.

3.2 Temporal reprojection

Ray marching optimizations reduce the render time considerably, but to get to acceptable render time figures it is necessary to reduce the number of pixels rendered each frame. The easiest solution is to simply lower the resolution leading to a drop in picture quality.



(a) Early exit, when reahing high opacity (b) Longer steps, when detail isn't needed

Figure 3.2: Ray marching optimizations

Another approach can be used to both maintain the picture quality and reduce the render time. Only some of the pixels will be rendered, while others can be copied from the previous frame.

In this implementation 1/16 of pixels are updated each frame. The picture is divided into 4x4 pixel blocks, where only one pixel is updated each frame. The order the pixels are updated in matters greatly. Bad update order results in patterns emerging in the clouds. An example of this is a naive order (Figure 3.3a). The updated pixels form vertical lines, which become visible in the render. The first order used in this thesis was $7n \mod 16$ (Figure 3.3b). The patterns are not as easily visible as in the naive order, but for example, 6 and 2 being close to each other becomes noticeable. An even better order can be described by the cross pattern (Figure 3.3c). The cross pattern is used in this thesis, as other tested orders (like random order) were more noticeable.

After applying this optimization the cloud rendering time becomes about 12 times faster (not 16 times, because some time is spent on reprojection).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a) Naive order

1	8	15	6
13	4	11	2
9	16	7	14
5	12	3	10

(b) Multiplications of seven

1	9	3	11
13	5	15	7
4	12	2	10
16	8	14	6

(c) Cross pattern

Figure 3.3: Orders for updating pixels

When the camera or the player moves quickly however, ghosting appears (Figure 3.4). This is because the pixels coppied from the previous frame haven't moved accordingly. These pixels need to be reprojected to an approximately correct position. Ghosting is then pushed to higher movement speeds.

The reprojection calculates the change in the eye ray direction towards the sample based on the movement of the camera and the clouds. Both the movement

of the camera is simply the change in its position between frames. The horizontal movement of the clouds is calculated from the wind speed, direction, and time passed between the frames. By approximating, that the sampled part of the cloud moves at the same speed as the cloud map, the change of the relative position Δp of the sample to the camera is easy to obtain. It is given by subtracting the camera movement from the cloud sample.

Calculating the new position of the old sample would be simple if its previous position would be known. However, the only information is given by the normalized direction from the camera towards the sample. The previous position could be calculated from this vector the vertical position of the sample was known. Because the vertical position of the sample is unknown, the goal is to pick the best estimate.

The clouds are usually the widest at the altitude, where their height based density function has the highest value. Depending on the type of cloud it can be between 725 and 1100 meters. The ghosting more visible at the edges of the clouds, than at its bottom. And the edges of the clouds are located in altitudes with high-density function values. In this thesis, 900 meters was picked as the best estimate. The estimated new position P of the sample is calculated from the eye vector ω and altitude estimate $h = 900m$ and relative position change Δp in the following equation:

$$P = \omega * 900/\omega_x + \Delta p \tag{3.1}$$

The corrected eye ray is then easy to compute. While the reprojection works best for the samples with altitude near 900 meters above the ground (most of the samples), it also reduces ghosting in other altitudes. Even the sample at 500 meters gets the offset caused by ghosting reduced by 55%.

Figure 3.4: Ghosting caused by rapid camera movement

Figure 3.5: The amount of ghosting can be reduced by reprojection

4. Results

4.1 Visual Results

4.1.1 Cumulus

Figure 4.1: Rendered cumulus clouds

4.1.2 Stratocumulus

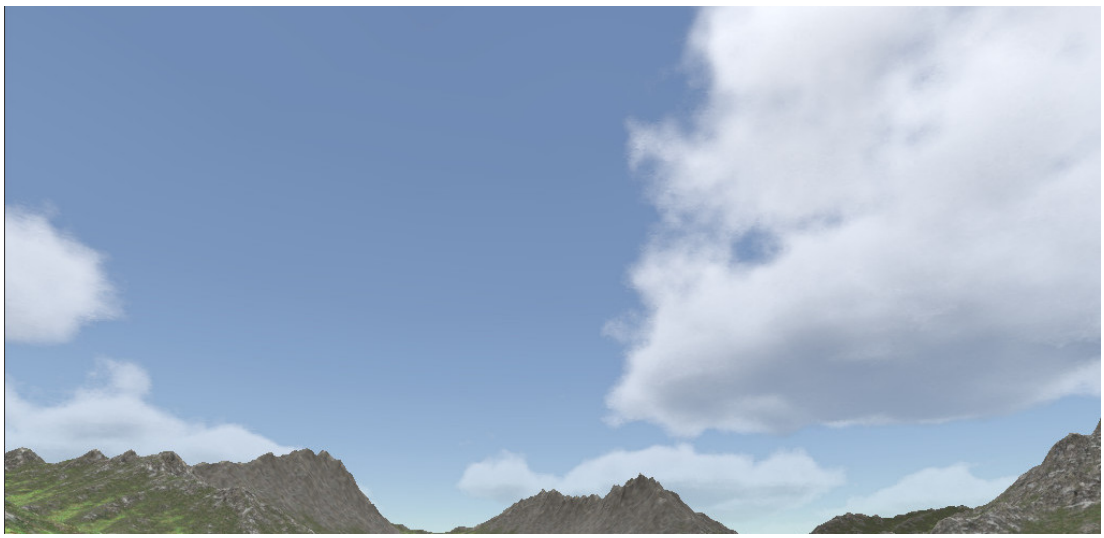


Figure 4.2: Rendered stratocumulus clouds

4.1.3 Stratus

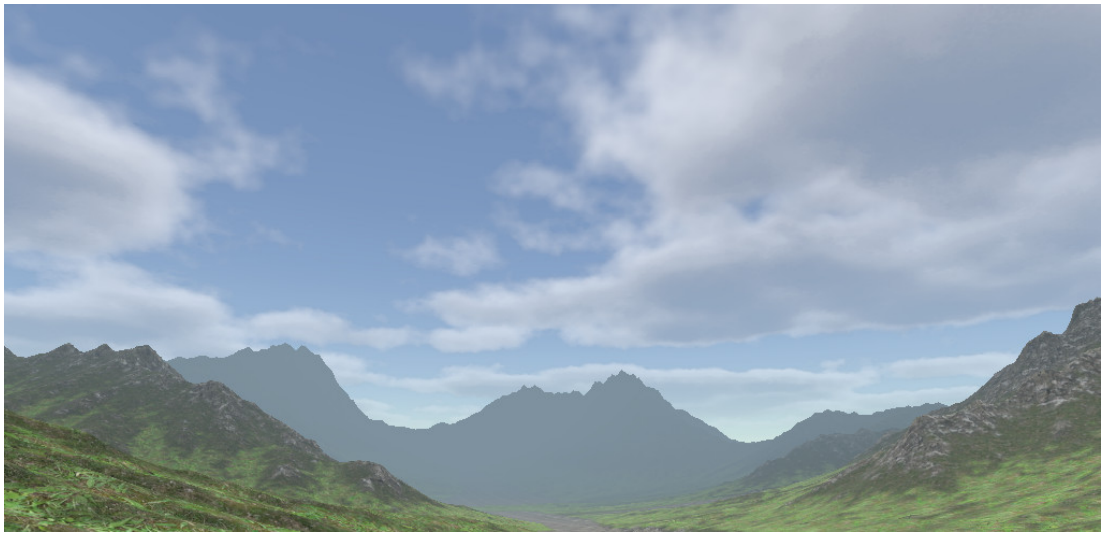


Figure 4.3: Rendered stratus clouds

4.1.4 Sunset

Figure 4.4: Rendered stratus clouds at sunset

4.2 Performance

The clouds were rendered at 2048x1024 cloud texture resolution on Windows 10 machine with Intel(R) Core(TM) i7-4700MQ processor, 16GB of DDR2 RAM, and GeForce GT 755M graphics card. Most of the computation is done in shaders, the GPU performance is by far the most important factor for render speed. The screen resolution does not impact the cloud rendering speed, as the rendering resolution is determined by the cloud texture.

The thesis follows the implementation of Schneider and Vos [14] for Horizon Zero Dawn, where clouds were rendered in less than 2ms. Radeon RX 480 graphics

card, which was recommended for this game, performs about 5 times faster on general rendering benchmarks than GeForce GT 755M. Clouds in this thesis should therefore be always rendered under 10ms to roughly match the Horizon Zero Dawn Performance.

The performance of the cloud rendering and generation was measured for different cloud types and coverages. Unity build-profiler was used for this purpose. When the profiler is used on GPU it produces some overhead. Real cloud rendering times might be therefore slightly lower than the measured ones. The results of the performance testing can be seen in Table **??**.

The Cloud map was generated in about 0.8ms, sunshafts postprocessing completed in 0.3ms, reprojection done in 1ms, and the speed of the cloud rendering itself varied based on the type of cloud and coverage. In general skies with lower coverage and thinner clouds were rendered faster.

Cloud Type	Coverage	Update Speed	Total Speed
stratus	0.3	2.05ms	4.15ms
	0.5	2.81ms	4.91ms
	1.0	3.98ms	6.08ms
stratocumulus	0.3	3.02ms	5.12ms
	0.5	3.86ms	5.96ms
	1.0	4.69ms	6.79ms
cumulus	0.3	3.34ms	5.44ms
	0.5	5.45ms	7.55ms
	1.0	6.31ms	8.41ms

Table 4.1: The table presents rendering performance measured for different cloud types and coverages. The Update Speed refers to the time it it takes to update 1/16 of the cloud texture. Total Speed describes the rendering time spent on rendering clouds, generating the cloud map and adding sun shafts.

Conclusion

The cloud rendering technique presented in this thesis achieved realtime performance on GeForce GT 755M, with render times ranging from 3 to 7 milliseconds (4 to 8 milliseconds, if cloud map generation and sun shaft post-process are included). Such render time is too costly for most computer games, target resolution can be lowered to achieve more suitable render time.

The cloud render speed was however tested on older hardware and for example, the recommended GPU for Horizon Zero Dawn (the game where the rendering system presented by Schneider was used) is about 5 times faster. If used on modern hardware, the cloud rendering system can be successfully used in computer games, not taking too much rendering time.

Most of the requirements of the games, where the player is positioned on the ground, were successfully implemented in this thesis. Multiple types of clouds are supported. Clouds change position and shape over time. Sky coverage can be set up by the user. Multiple times of day can be simulated. A dynamically changing cloud map with multiple cloud types can be generated. The program is however unable to animate user-supplied cloud maps.

Clouds in this thesis are rendered as projections on the skybox. This makes it impossible to fly through the clouds or place objects in them. The system is therefore unsuitable for plane simulators, or other games, where the player is positioned near, or above the clouds. Clouds for example also not cover mountains, or any objects for that matter.

Future work

The main current limitation of the *JK Volumetric Clouds for Unity* are the cloud maps. The cloud map supplied by the user as a static 2D texture does not change. This means, that the position of the clouds generated by this map can not be modified, and neither can the sky coverage. A function could be provided, that changes the user-defined cloud map each frame and generates new parts of the map once the clouds created by the user move out of the view. These newly generated parts of the map must be similar to the original clouds.

The generated cloud map supports different cloud types changing their shape and position over time. The type of each cloud is assigned randomly. The cloud types could be organized more logically. All of the clouds also move their position in the same direction and speed in the direction of the wind. Weather effects like cyclones, where the clouds move in a more complicated manner could be simulated.

It may be possible to improve performance in several ways. Temporal reprojection applied more aggressively would speed up the rendering speed, but increase ghosting. This could be used when the cloud position does not change too quickly. Raymarching with length chosen smartly for each of the steps could reduce the required number of steps while maintaining the visual quality. If the clouds do not change their shape over time, the lighting can be precomputed increasing rendering speed significantly.

Improvement outside of the scope of this thesis would be connecting the volumetric clouds to a separate weather system. Incoming clouds could be for example connected to rain or storms.

Bibliography

[1] 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*, pages 173–182, 2008.

[2] Eric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. *Computer Graphics Forum*, 27(4):1079–1086, 2008.

[3] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[4] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-Time Volume Graphics*. A. K. Peters, Ltd., USA, 2006.

[5] Sébastien Hillaire. Physically based sky, atmosphere and cloud rendering in frostbite, 2016. Online, Accessed: 2020-5-18.

[6] Joe Kniss, S. Premoze, C. Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. pages 109–116, 12 2002.

[7] Met Office. Clouds. Online, Accessed: 2020-5-18.

[8] Ksenia Mukhina and Alexey Bezgodov. The method for real-time cloud rendering. *Procedia Computer Science*, 66:697–704, 12 2015.

[9] Adolfo Muñoz. Higher order ray marching. *Computer Graphics Forum*, 33(8):167–176, 2014.

[10] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[11] Jan Novák, Iliyan Georgiev, Johannes Hanika, Jaroslav Křivánek, and Wojciech Jarosz. Monte carlo methods for physically based volume rendering. In *ACM SIGGRAPH 2018 Courses*, SIGGRAPH '18, pages 14:1–14:1, New York, NY, USA, 2018. ACM.

[12] Ken Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296, New York, NY, USA, 1985. Association for Computing Machinery.

[13] Timothy Roden and Ian Parberry. Clouds and stars: efficient real-time procedural sky rendering using 3d hardware. pages 434–437, 01 2005.

[14] Andrew Schneider and Nathan Vos. The real-time volumetric cloudscapes of horizon zero dawn, 2015. Online, Accessed: 2020-5-18.

[15] Andrew Schneider and Nathan Vos. Nubis: Authoring real-time volumetric cloudscapes with the decima engine, 2017. Online, Accessed: 2020-5-18.

[16] Sea of Thieves. Creating clouds, 2016. Online, Accessed: 2020-5-19.

[17] Stanley S Stevens. On the psychophysical law. *Psychological review*, 64(3):153, 1957.

[18] World Meteorological Organization. Classifying clouds, 2017. Online, Accessed: 2020-5-18.

[19] World Meteorological Organization. International cloud atlas, 2017. Online, Accessed: 2020-6-2.

[20] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 291–294, New York, NY, USA, 1996. Association for Computing Machinery.

[21] E. Yusov. High-performance rendering of realistic cumulus clouds using precomputed lighting. *High-Performance Graphics 2014, HPG 2014 - Proceedings*, pages 127–136, 01 2014.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment