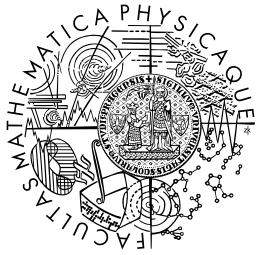


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

In video games, the sky often covers close to half of the player's 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.

0.1 Clouds in computer games

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, 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 can not be seamlessly changed, clouds can not move based on the wind or the player position. There are several approaches to solve these issues.

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 (Roden and Parberry [2005]). This allows for a dynamic change of coverage and shape of the clouds. However, the visual results usually look flat and unconvincing.

Representing clouds as 3D objects can solve the problems of the previously mentioned methods. Clouds can be visualized using volumetric rendering. This usually means raymarching through the volumetric data and taking samples of density and lighting each step. Volumetric clouds used to be too computationally heavy for use in computer games because the sampling of 3D data requires a large number of texture reads. Recent advances in the consumer hardware performance, however, made their use possible.

0.2 Related work

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 (Schneider and Vos [2015]). Schneider continued his work and the current cloud system is called Nubis (Schneider and Vos [2017]). It achieved several improvements, especially in the lighting.

Hillaire [2016] offer a solution similar to Schneider's work and achieve some improvements in lighting.

Ebert et al. [2002] models volumetric clouds by perturbating implicit volumes.

Yusov [2014] creates clouds from smaller particles, each with specified radius, color and density.

Mukhina and Bezgodov [2015] generate clouds procedurally in 2D. Using advanced lighting simulations, they manage to generate realistic clouds evolving over time.

0.3 Goals

The aim of the thesis is to implement realistic real-time volumetric clouds similar to Schneider's work. 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 must 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

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 Clouds

The information used in this section was taken from Met Office.

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.

1.1.1 Cloud Types

The information in this section was taken from Met Office and the World Meteorological Organization [2017]

Not all clouds are the same, they differ in their shape, size, and altitude. The International Cloud Atlas 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)

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) is 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.

https://tinyurl.com/y92fc7jx

(a) Stratus

https://tinyurl.com/y6wrs99h

(b) Cumulus

https://tinyurl.com/y99gfggf

(c) Cumulonimbus

https://tinyurl.com/y8ysr34p

(d) Stratocumulus

Figure 1.1: Low altitude clouds

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.

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.

https://tinyurl.com/y92j2dkv

(a) Altostratus

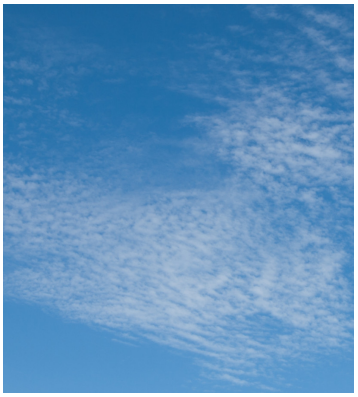
https://tinyurl.com/y7tkfbht

(b) Nimbostratus

https://tinyurl.com/y96qpzko

(c) Altocumulus

Figure 1.2: Medium altitude clouds



https://tinyurl.com/yb8ml518

(a) Cirrocumulus

https://tinyurl.com/y7w24sad

(b) Cirrostratus

https://tinyurl.com/y9uy4ok8

(c) Cirrus

Figure 1.3: High altitude clouds

1.1.2 Lighting of Clouds

When the light enters the cloud it interacts with the water droplets and ice crystals the cloud in several ways. It can be absorbed, or scattered by these particles.

Absorption

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

$$\boldsymbol{A = \epsilon c l}$$

A	Absorbtion	
$\boldsymbol{\epsilon}$	Molar absorption coefficient	$M^{-1}m^{-1}$
c	Molar concentration	M
l	Length of the optical path	m

Absorbance isn't very useful to describe what part of light was absorbed, transmittance $\boldsymbol{T}$ should be used for this purpose.

$$\boldsymbol{A = -log_{10} T} \tag{1.1}$$

$$\boldsymbol{T} = \boldsymbol{e}^{-\beta_a l} \tag{1.2}$$

Where $\boldsymbol{\beta_a}$ is an absorbtion coefficient combining both $\boldsymbol{\epsilon}$ and $\mathbf{c}$ from the original Beer-Lambert law.

This means, that the amount of light transmitted through the cloud decreases exponentially with distance. The previous equation however only describes light transmission in a homogeneous medium. Gass density in real clouds varies and different equation is necessary to describe transmittance in nonhomogeneous medium. Equation (1.3) calculates it between the points $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$, where $\boldsymbol{\beta_a(s)}$ is an absorbtion coefficient at the point $\boldsymbol{s}$.

$$\boldsymbol{T} = \boldsymbol{e}^{-\int_{s1}^{s2} \beta_a(s) ds} \tag{1.3}$$

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, this can be seen in Figure 1.4.

https://bit.ly/2TyOn5K

Figure 1.4: Self shadowing in clouds

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.

Scattering in the clouds is anisotropic, it is more likely for the scattered light to go in a direction similar to the original light direction. As a resul, clouds closer to the Sun dot have silver linings and are generally lighter (see Figure 1.5a)

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 (1.4).

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

Absorption and out-scattering coefficiens are be combined into a single extinction coefficient in Equation (1.5).

$$\boldsymbol{\beta_e = \beta_a + \beta_s} \tag{1.5}$$

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

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 1.5b).

https://tinyurl.com/y7k8mpth

(a) Silver lining near the Sun dot

https://tinyurl.com/ycwyx4d2

(b) Dark edges of the clouds

Figure 1.5: Cloud lighting effects

1.2 Cloud Modeling

The model is a representation of an object used by the computer. Usually, the geometry of the model is described by a set of points grouped into polygons. This approach can also be used to represent the clouds Sea of Thieves [2016]. 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.

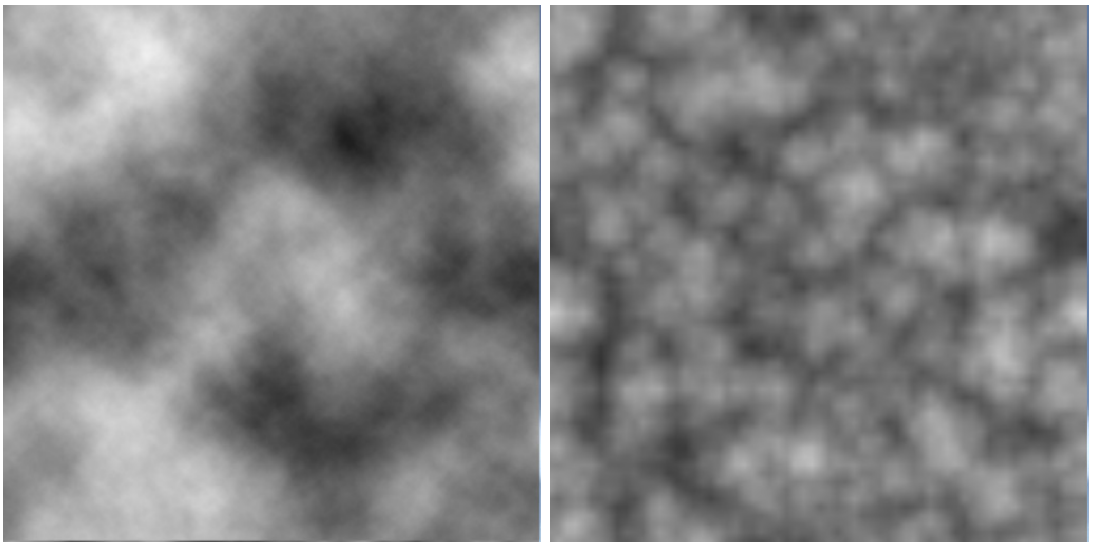
Clouds can also be represented by particles, each represented by position, radius, density, and color Yusov [2014].

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. [2002] and Ebert et al. [2002] 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 and Vos [2015] 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 noise (Figure 1.6b) and a layered Perlin noise (Figure 1.6a). The detail is added by another 3D layered Worley noise texture.



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

Figure 1.6: Noises used for cloud modeling

1.3 Cloud Rendering

Rendering in computer graphics means generating an image from the model. 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 (1.7) (as shown by Novák et al. [2018]).

$$\boldsymbol{L}(\boldsymbol{x},\boldsymbol{\omega}) = \int_{\mathbf{0}}^{\boldsymbol{z}} \boldsymbol{T}(\boldsymbol{x},\boldsymbol{y})[\boldsymbol{\sigma}_a(\boldsymbol{y})\boldsymbol{L}_e(\boldsymbol{y},\boldsymbol{\omega}) + \boldsymbol{\sigma}_s(\boldsymbol{y})\boldsymbol{L}_s(\boldsymbol{y},\boldsymbol{\omega})]\boldsymbol{dy} + \boldsymbol{T}(\boldsymbol{x},\boldsymbol{z})\boldsymbol{L}_o(\boldsymbol{z},\boldsymbol{\omega}) \tag{1.7}$$

Where $\boldsymbol{T(x,y)}$ describes the transmittance between the point $\boldsymbol{x}$ and $\boldsymbol{y}$ (the fraction of light from the point $\boldsymbol{y}$, that reaches the point $\boldsymbol{x}$). The transmittance can be calculated using the Equation (1.8).

$$\boldsymbol{T(x,y) = e^{-\int_0^y \beta_e(s)ds}} \tag{1.8}$$

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

$$\boldsymbol{\sigma_a(y)L_e(y,\omega) = 0} \tag{1.9}$$

$\boldsymbol{\sigma_s(y)L_s(y,\omega)}$ describes the in-scattering (light scattering towards the eye). The amount of in-scattered light can be computed using the Equation (1.10).

$$\boldsymbol{L_s(y,\omega) = \int_{S^2} f_p(\omega,\omega_2)L(y,\omega_2)d\omega_2} \tag{1.10}$$

Where $\boldsymbol{f_p(\omega,\omega_2)}$ is a phase function describing the probability of the light from direction $\boldsymbol{\omega_2}$ to scatter in the direction $\boldsymbol{\omega}$. Cloud particles tend to scatter the light forward (in a direction similar to the original). Many different functions can be utilized depending on the volume characteristics. For larger particles, the clouds are composed of, the using the Mie scattering theory produces the most accurate phase functions. They are usually diffucult to work with because of thier complexity. Simpler functions give approximations sufficient for real time applications. Hadwiger et al. [2006] suggests using renormalized Heyney-Greenstein phase function (1.11).

$$\boldsymbol{f_p(\omega,\omega_2) = \frac{1-g^2}{(1+g^2-2g\frac{\omega\cdot\omega_2}{|\omega|\cdot|\omega_2|})^{3/2}}} \tag{1.11}$$

$\boldsymbol{L_o(z,\omega)}$ describes the amount of light directed in the eye direction by the object behing the cloud volume.

The volume rendering equation is recursive, because $\boldsymbol{L(y,\omega)}$ 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 (Bouthors et al. [2008]). 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 using ray marching and single scattering.

Ray marching is a popular method of rendering volumetric data (Muñoz [2014]). 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 Bouthors et al. [2008]

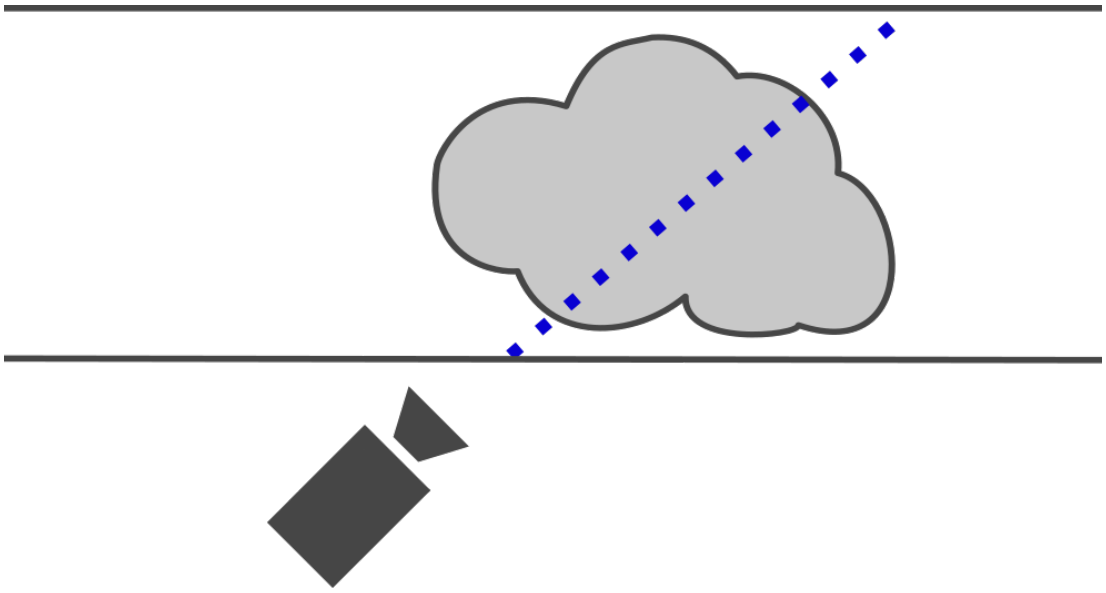


Figure 1.8: Ray marching visualization (samples - blue dots)

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 HLSL language was used for shaders and C# language for scripts.

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.

Simplified process of the cloud shape definition

1. A 2D texture called weather 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.

2.1.1 Cloud map

The weather map is a 2D texture, that controls the position, type, amount, and density of the clouds. The resolution used in this implementation is 512x512.

The weather map 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 higher value in the channel gives a higher probability of clouds of the selected type appearing. Cloud thickness and density are also increased. The alpha channel stores cloud coverage, a variable used in automatic cloud map generation.

Two types of cloud maps can be distinguished.

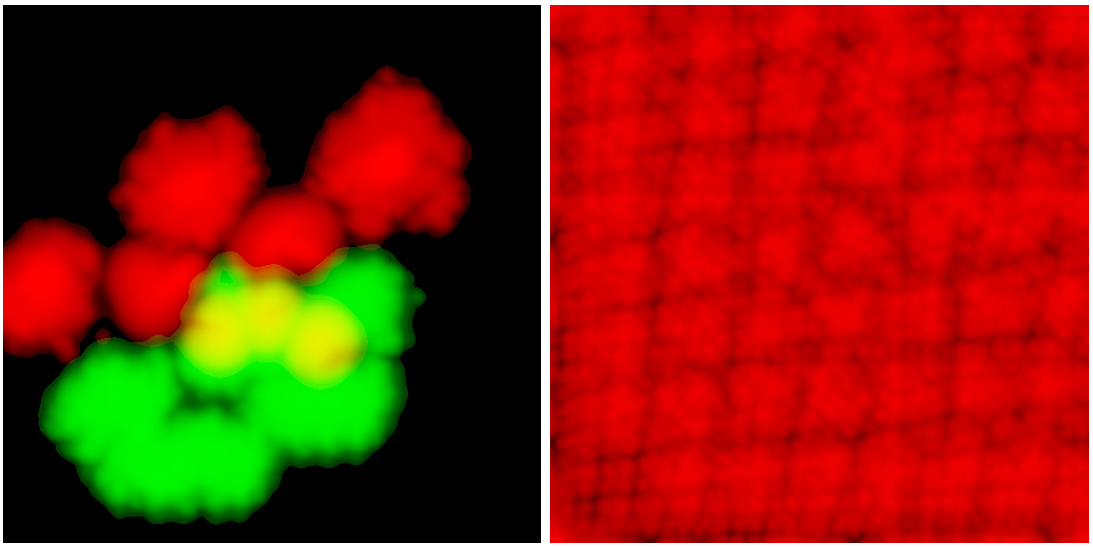
User-supplied

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 2.1a.

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. Only cloud maps with one cloud type can be generated automatically by the program. This cloud type can be picked by the user. An example of an automatically generated cloud map can be seen in Figure 2.1.

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 variable. The coverage of the clouds newly appearing on the horizon will slowly move closer to the desired value.



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

Figure 2.1: Examples of cloud maps

2.1.2 Height dependent shape altering functions

The program renders all clouds between 500 and 2000 meters, 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.

The 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 2.2.

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

Figure 2.2: 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 2.3.

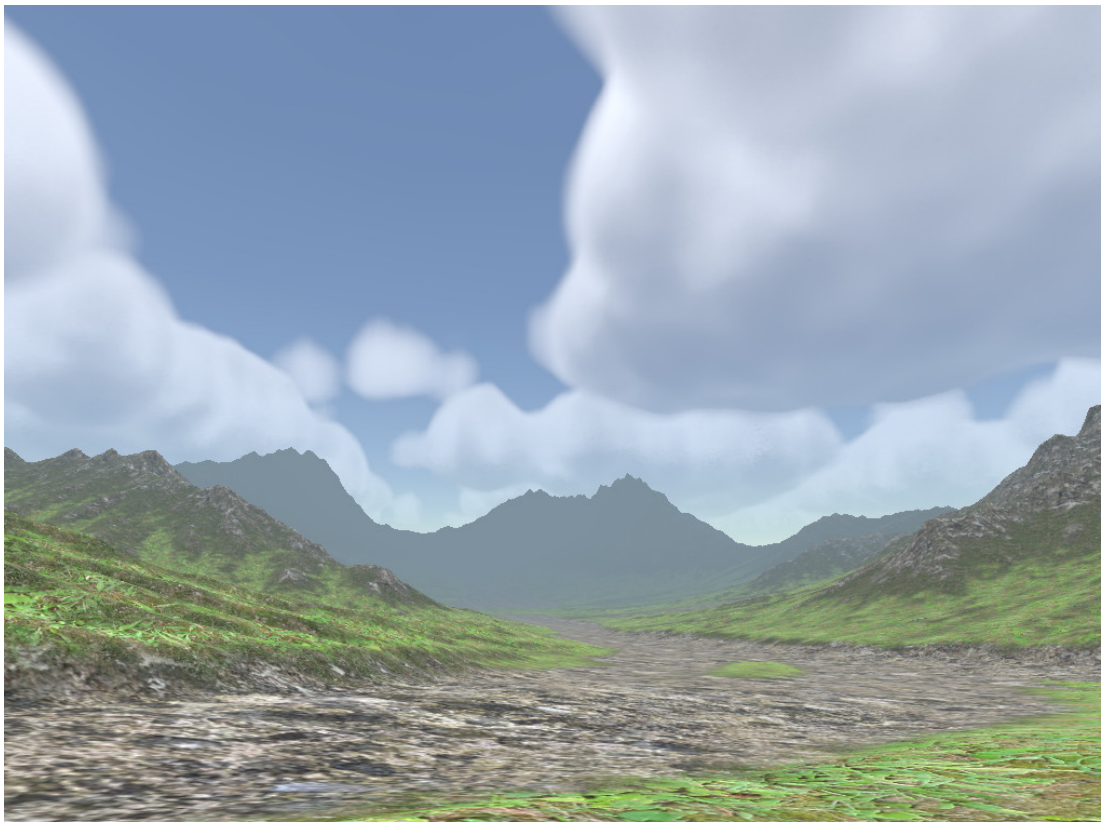


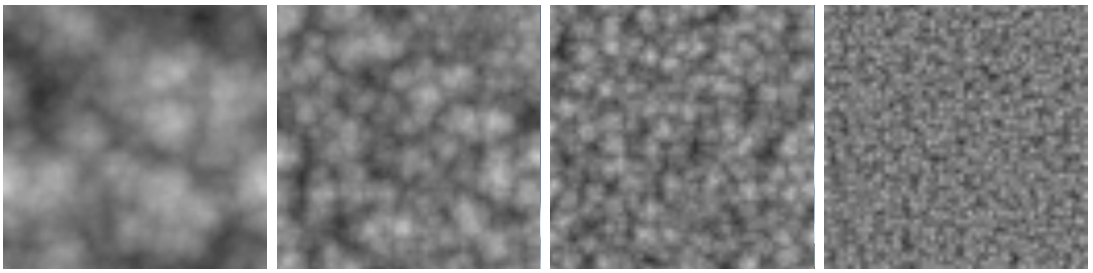
Figure 2.3: Shape of the clouds defined by the cloud map and the shape altering height function

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 2.4.



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

Figure 2.4: 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, $\boldsymbol{s_r}$ is the noise from the red color channel, $\boldsymbol{s_g}$ noise from the green channel, $\boldsymbol{s_b}$ noise from the blue channel, and $\boldsymbol{s_a}$ from the alpha channel.

$$\boldsymbol{SN_{sample} = 0.6s_r + 0.6s_g + 1.0s_b + 0.1s_a} \tag{2.1}$$

The shape noise sample is then combined with the weather map and the height dependent shape altering function in (2.2) (0.4 is treshold for the clouds to appear picked as an artistic decision).

$$\boldsymbol{Sample = SN_{sample} \cdot CloudMap \cdot HeightFunction - 0.4} \tag{2.2}$$

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 2.5.

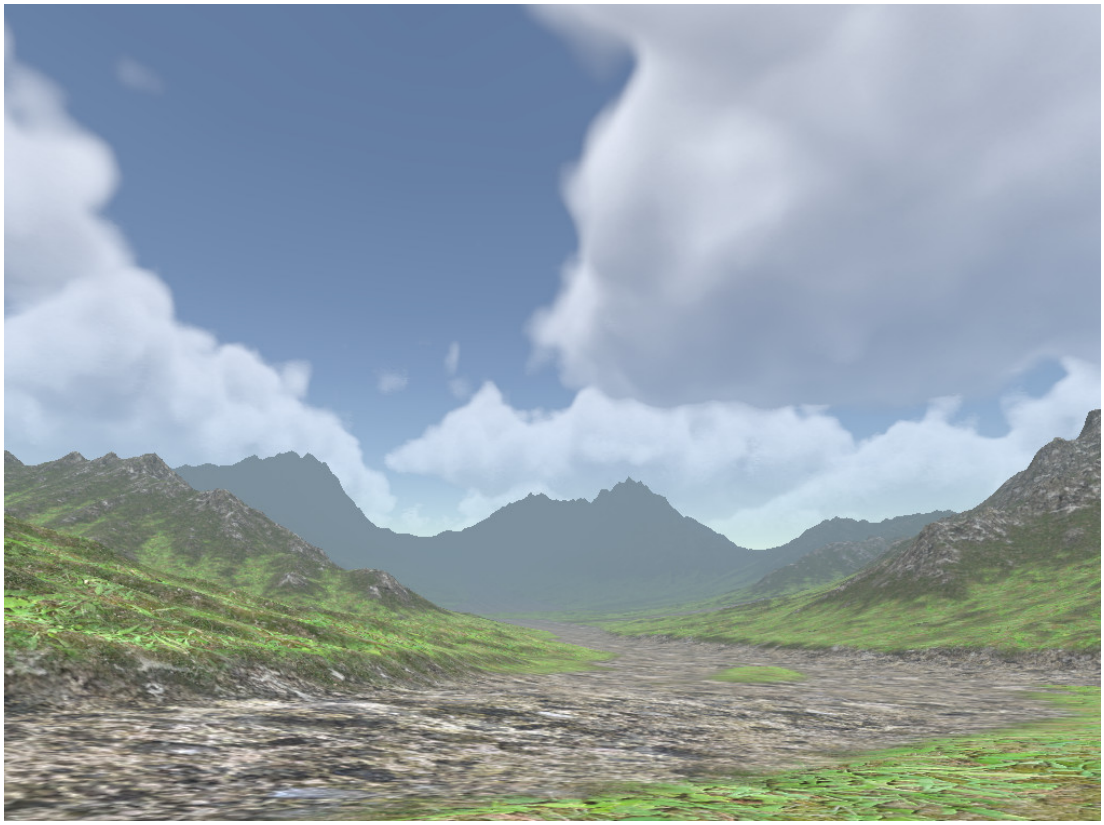


Figure 2.5: Clouds after applying shape noise

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 to reduce possible artifacts. The result can be seen in Figure 2.6

Figure 2.6: Clouds after applying detail noise

2.1.5 Cloud movement

Clouds are moved across the sky by offsetting the cloud map (only when generated), shape noise, and detail noise based on the direction and speed of the wind. Each texture is moved at a different speed, which makes the cloud shapes evolve.

The user-supplied cloud map is not offset by the wind, because there is no way to tell, how the newly appearing areas should be generated. Clouds generated using this map do not move over the sky, but they still change shape based on the movement of the 3D noise textures.

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.

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.

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

Non-real-time 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.

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

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.

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

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).

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

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

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

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.

Raymarching offset

When raymarching starts at the same height for each point in the sky, a banding pattern appears (Figure 2.7a). The problem can be solved by offsetting the start of each raymarching by a pseudorandom distance (Figure 2.7b).

(a) No offset (b) With offset

Figure 2.7: 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.

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

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

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

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).

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

Beer-Lambert law

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

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

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.

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.

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

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 2.8.

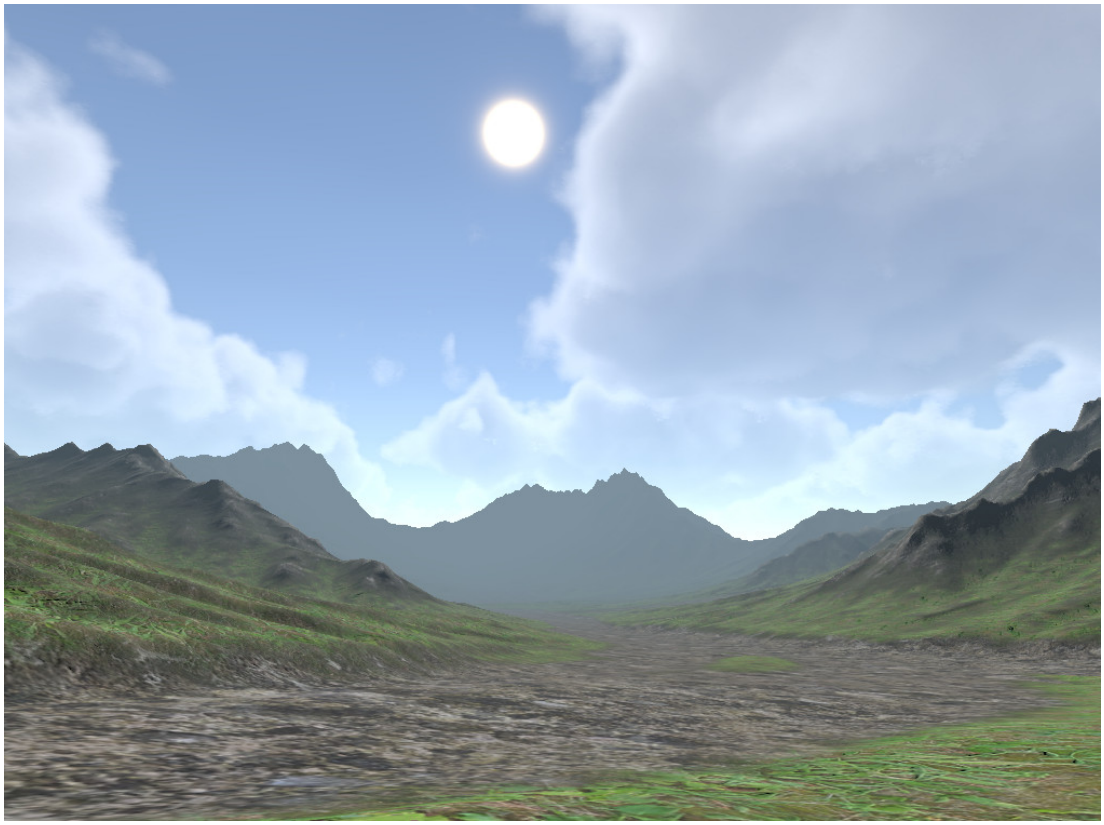


Figure 2.8: Cloud self shadowing after applying the Beer-Lambert law

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) with parameter $\boldsymbol{g = 0.7}$. A positive value of $\boldsymbol{g}$ means, that the light tends to scatter forward.

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

After applying the Heyney-Greenstein phase function silver linings around the Sun dot appear (see Figure 2.9).

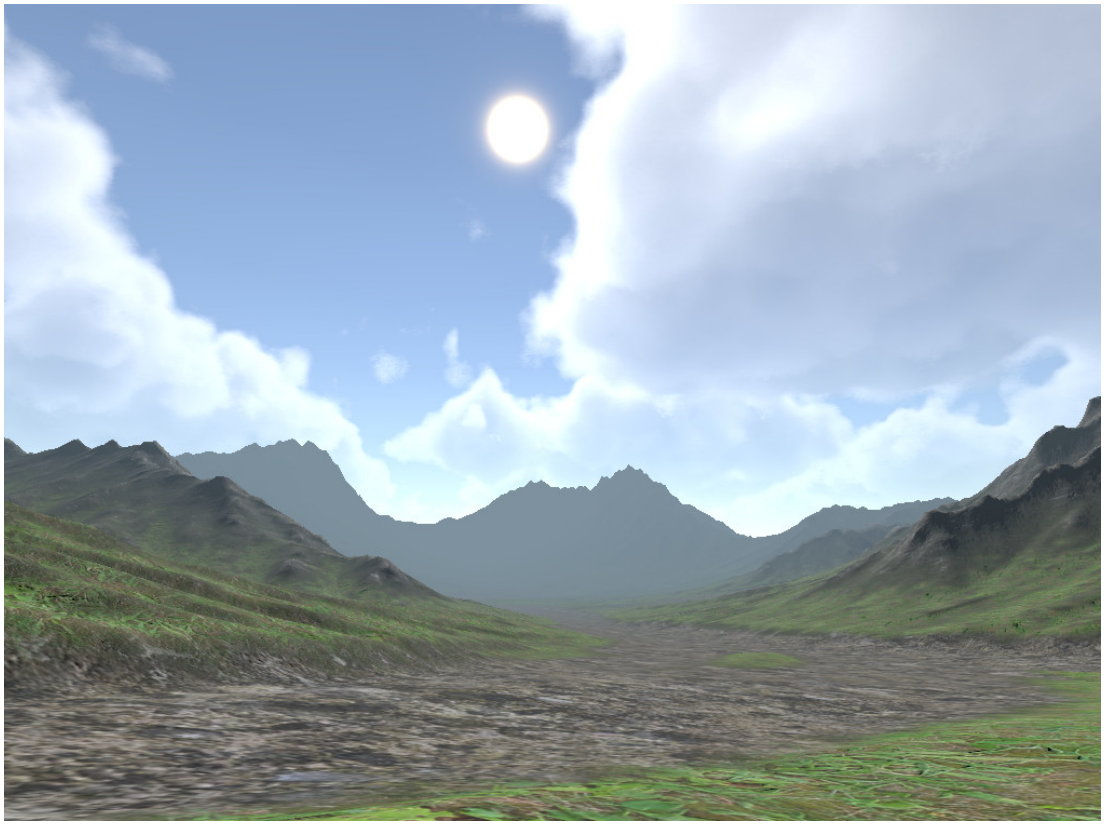


Figure 2.9: Clouds after applying the Heyney-Greenstein phase function

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. 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 2.10 shows different sky coverages casting shadows.

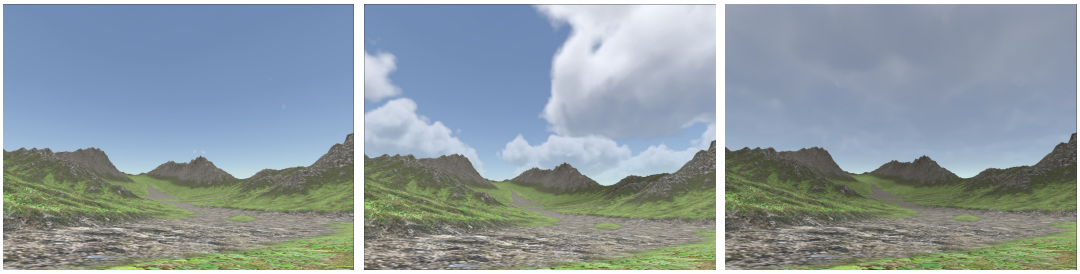


Figure 2.10: Shadows cast by different sky coverages

Sunshafts

Sunshafts are implemented as a post-processing effect described by Nguyen [2007]. These Sun shafts not exactly correspond to their real-world counterpart, but they are faster to render than truly volumetric sunshafts.

The shaft rendering consist of three steps. First unobstructed light areas of the skybox are selected (Figure 2.11a), these are then blurred in the direction away from the Sun dot (Figure 2.11b). Finally the blurred image is then laid over the originally rendered image (Figure 2.11c).

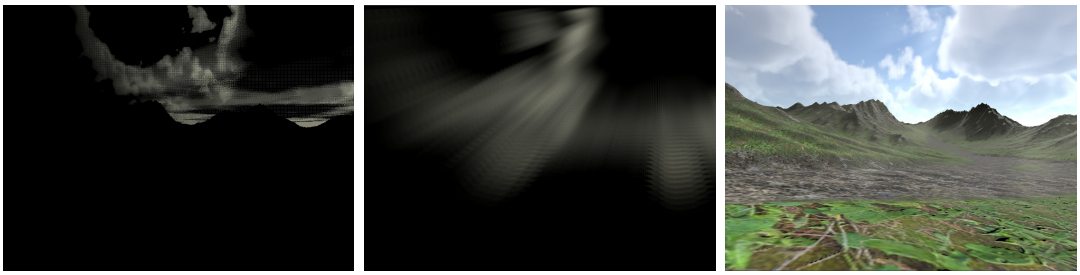


Figure 2.11: Sunshafts generation process

Ambient

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). Implementation used in this thesis lineary interpolates between two ambient light color based on the altitude of the sampled point.

2.2.3 Atmosphere blending

To create a realistic scenery atmosphere must be taken into account. Adding blending clouds with the atmosphere gives clouds a more realistic look and a sense of depth. The color of the atmosphere is the same as the color of the underlying skybox. 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 2.12.

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

Figure 2.12: 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.

Skipping the shape and detail noise makes the shader significantly faster. Two out of the three texture samplings are skipped.

Rendering of the sky in Figure 3.1 is sped up considerably by this optimization. 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.

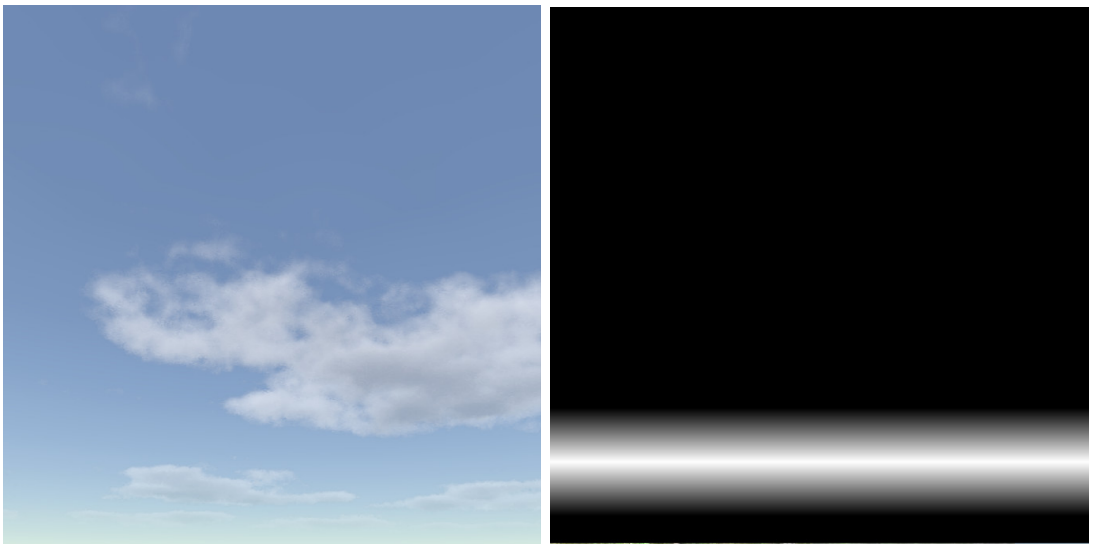


Figure 3.1: Sky optimized by skipping of the 3D texture sampling

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 detail noise should be only sampled, when the density obtained from the combination of cloud map, height function, and shape noise is close to zero. 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.

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.

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.

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 pixels are updated in an order specified by the cross pattern, which reduces a chance of aliasing and visible patterns appearing in the sky.

When the camera or the player moves quickly however, ghosting appears (Figure 3.2). 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.

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

Figure 3.2: Ghosting caused by rapid camera movement

4. Results

4.1 Visual Results

4.1.1 Cumulus

Figure 4.1: Rendered cumulus clouds

4.1.2 Stratocumulus

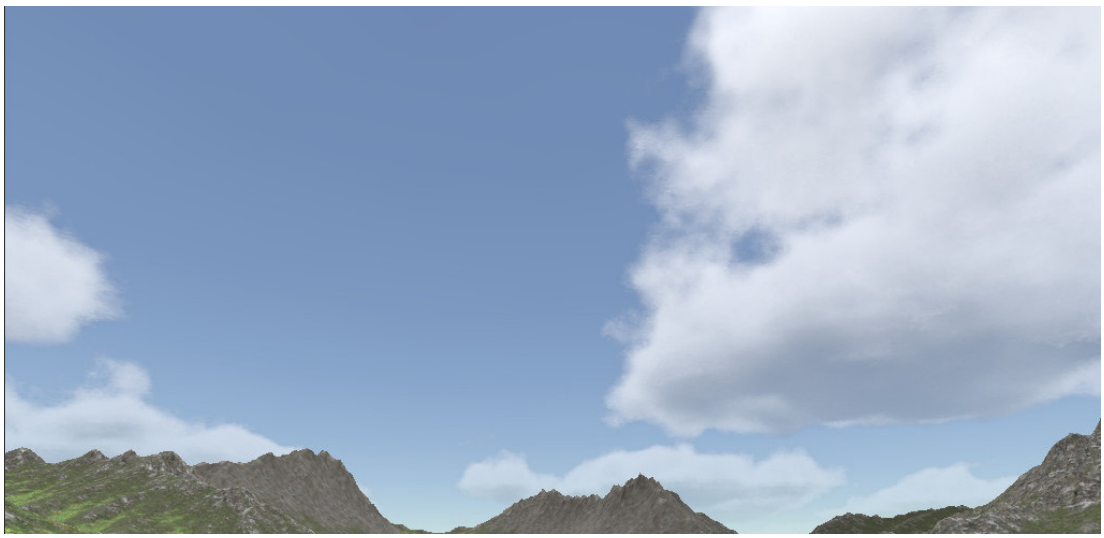


Figure 4.2: Rendered stratocumulus clouds

4.1.3 Stratus

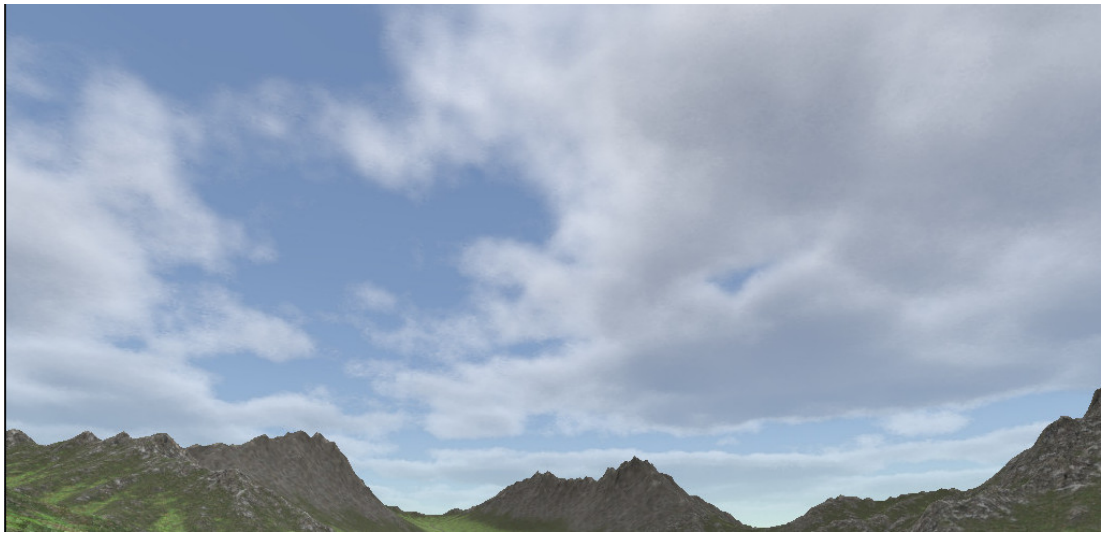


Figure 4.3: Rendered stratus clouds

4.1.4 Sunset

Figure 4.4: Rendered stratus clouds at sunset

4.1.5 Night

Figure 4.5: Rendered cumulus clouds in the night

4.2 Performance

The clouds were rendered at 2048x1024 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 thesis follows the implementation of Schneider and Vos [2015] 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 be always rendered under 10ms.

The performance was measured using Unity in-build profiler. 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 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
stratus	0.3	2.05ms
stratus	0.5	2.81ms
stratus	1.0	3.98ms
stratocumulus	0.3	3.02ms
stratocumulus	0.5	3.86ms
stratocumulus	1.0	4.69ms
cumulus	0.3	3.34ms
cumulus	0.5	5.45ms
cumulus	1.0	6.31ms

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. The program, however, can not automatically generate a cloud map for a sky with multiple types of clouds and it is 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 problem the JK Volumetric Clouds for Unity has are the cloud maps. The cloud map supplied by the user as a 2D texture can not change. This means, that the position of the clouds generated by this map does not change and neither does the sky coverage. Automatically generated cloud maps can do both of these things, but only one type of cloud can be present in the cloud map at a single time.

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.

Having different cloud types in the generated cloud map is quite easy. Each Worley cell could have a cloud type assigned. Clouds of each type are then distributed randomly. The more complicated part is having the clouds organized more logically.

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 a raymarching step 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

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.

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. ISBN 1558608486.

Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-Time Volume Graphics*. A. K. Peters, Ltd., USA, 2006. ISBN 1568812663.

Sébastien Hillaire. Physically based sky, atmosphere and cloud rendering in frostbite, 2016. URL `https://media.contentapi.ea.com/content/dam/eacom/frostbite/files/s2016-pbs-frostbite-sky-clouds-new.pdf`. Online, Accessed: 2020-5-18.

Joe Kniss, S. Premoze, C. Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. pages 109–116, 12 2002. ISBN 0-7803-7498-3. doi: 10.1109/VISUAL.2002.1183764.

Met Office. Clouds. URL `https://www.metoffice.gov.uk/weather/learn-about/weather/types-of-weather/clouds`. Online, Accessed: 2020-5-18.

K. Mukhina and A. Bezgodov. The method for real-time cloud rendering. *Procedia Comp. Science*, 66(2):697–704, 2015.

Adolfo Muñoz. Higher order ray marching. *Computer Graphics Forum*, 33(8):167–176, 2014. doi: 10.1111/cgf.12424. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12424`.

Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428.

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. ISBN 978-1-4503-5809-5. doi: 10.1145/3214834.3214880. URL `http://doi.acm.org/10.1145/3214834.3214880`.

Timothy Roden and Ian Parberry. Clouds and stars: efficient real-time procedural sky rendering using 3d hardware. pages 434–437, 01 2005. doi: 10.1145/1178477.1178574.

Andrew Schneider and Nathan Vos. The real-time volumetric cloudscapes of horizon zero dawn, 2015. URL `https://d1z4o56rleaq4j.cloudfront.net/downloads/assets/The-Real-time-Volumetric-Cloudscapes-of-Horizon-Zero-Dawn.pdf`. Online, Accessed: 2020-5-18.

Andrew Schneider and Nathan Vos. Nubis: Authoring real-time volumetric cloudscapes with the decima engine, 2017. URL https://d1z4o56rleaq4j.cloudfront.net/downloads/assets/Nubis-Authoring-Realtime-Volumetric-Cloudscapes-with-the-Decima-Engine-Final.pdf. Online, Accessed: 2020-5-18.

Sea of Thieves. Creating clouds, 2016. URL https://www.seaofthieves.com/it/news/short-haul-3. Online, Accessed: 2020-5-19.

World Meteorological Organization. Classifying clouds, 2017. URL https://public.wmo.int/en/WorldMetDay2017/classifying-clouds. Online, Accessed: 2020-5-18.

E. Yusov. High-performance rendering of realistic cumulus clouds using precomputed lighting. *High-Performance Graphics 2014, HPG 2014 - Proceedings*, pages 127–136, 01 2014. doi: 10.2312/hpg.20141101.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment