MASTER'S THESIS | LUND UNIVERSITY 2016

# Real-Time Rendering of Volumetric Clouds

Rikard Olajos

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-42

# Real-Time Rendering of Volumetric Clouds

(Ray Marching Noise Based 3D Textures)

Rikard Olajos
lat11rol@student.lu.se
clouds@rikardolajos.se

October 4, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Michael Doggett, michael.doggett@cs.lth.se
Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

## Abstract

Rendering volumetric clouds is a compute-intensive process which makes it difficult to use in real-time applications. At the same time, the need for volumetric clouds is evident as game developers look for new places to increase the realism of their games. Skyboxes and flat textures work well in 3D scenes where the camera is expected to be far away from the clouds and not move over large distances. But in open world games where the position of the camera cannot be assumed, skyboxes give a static impression and flat texture can give artefacts.

This thesis explores different techniques to save computational time when implementing volumetric clouds for real-time rendering. We start from a realistic implementation and from there propose different approximations and methods to see which performance gains can be accomplished, and at what costs.

Our implementation presents a way of forming cloud textures that uses a mixture of precalculation and real-time calculations, and allows for easy configurations and flexibility in creating different cloudscapes. For the cloud rendering, we present a way to preprocess the cloud texture and create a low resolution structure which saves a lot of rendering time.



**Keywords**: Clouds, Real-time rendering, Volumetric ray marching, Cellular noise, Deferred shading

# Acknowledgements

Firstly, I would like to thank my supervisor, Ass. Prof. Michael Doggett for always taking the time to assist and really understanding the challenges that arose during this thesis. I would also like to acknowledge Johan Sunnanväder and Madeleine Håkansson for their help in proofreading my work.

# Contents

# Chapter 1
# Introduction

Game developers are always pushing their games to be the most realistic in some aspect. Realistic and impressive graphics has always been such an aspect. But they can only push the graphics so far before computational power becomes an issue. This master thesis will investigate this balance between visual and computation performance and specifically it will investigate different approaches to generating volumetric clouds and cloudscapes. Skyboxes, using cube mapping, has been common practise in the industry for a long time for rendering clouds in the 3D world. This means that images of sky, clouds, and even distant scenery is placed at a locked distance around the viewer, giving the impression of an environment infinitely far away. The illusion of a sky surrounding the observer is held for scenes were the observer is placed close to ground level and is not expected to travel too far, such as first-person shooters and racing games. But in the increasingly popular open-world games and in flight simulators, a skybox would give a static feel and sense of travel would be lost. There is also no possibility to render the scene close to or inside of the clouds when using a skybox, which might be desirable. To remedy this, physical clouds must be placed in the 3D scene. This can be done using 2D texture but the results will likely look flat and not very realistic; volumetric clouds are needed for good results.

The goal of this master thesis is to produce volumetric clouds and explore different techniques and approaches to, under given circumstances, render them in real-time. *Real-time* means that we are aiming for a frame time of about 33 ms—effectively giving us a render frequency of 30 frames per second (FPS)[1]. We will start by implementing a model that mirrors what happens in real clouds and from there make trade-offs and algorithmic choices to achieve the wanted frame time. The frame time goal is of course arbitrary and the algorithms would need to be re-tuned for a real-world application. Even though the outcome would differ, the algorithms discussed in this thesis could still be used.

In Figures 1.1 and 1.2, photographs of clouds are presented to show the kind of effects we are hoping to accomplish. Figure 1.1 shows the kind of cloud form we are after as well

[1] Akenine-Möller et al. [1] defines *real-time* as being at least 15 FPS, but to make sure that the image is not perceived as too choppy we aim for 30 FPS and game developers often aim for 30–60 FPS.

as the lighting effect where the light gets attenuated as it passes through the clouds. The other figure, Figure 1.2, shows how the sun light scatters through the clouds, which results in bright edges and dark cores of the clouds, with the exception of the clouds covering the sun which are brightly lit.

**Figure 1.1:** A photograph showing clouds with rounded tops and flat bottoms. We can see how the light attenuates as it travels through the clouds.

**Figure 1.2:** A photograph showing how the light in the direction of the sun (behind the cloud in the centre of the photograph) has a higher amount of forward scattering.

The rest of this chapter will explain the underlying theory such as the rendering pipeline

and how it is accessed. At the end of this chapter some theory about clouds will be discussed as well as earlier works and contributions.

## 1.1 Rendering Pipeline

The rendering pipeline is the process in which we take the information described in our application and get it presented on our display, in other words the process of presenting the 3D space on the 2D display. Typically the rendering pipeline is described as in Figure 1.3. In the explanation throughout this chapter we will consider the modern type of rendering pipeline with programmable stages.

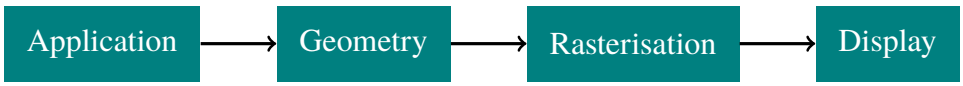



**Figure 1.3:** A very basic outline of the rendering pipeline. Application is were the 3D world resides and the Display is what the end user sees. All stages are explained in more detail below. Source: [14].

### 1.1.1 Application Stage

In the application stage, the scene to be rendered is built. It can consist of models, cameras, lights, etc. Typically these are described with a position in the world space and some kind of orientation. Take a cube for example. We can describe it as eight points $(0,0,0)$, $(1,0,0)$, $(1,1,0)$, $(0,1,0)$, $(0,0,1)$, $(0,1,1)$, $(1,0,1)$ and $(1,1,1)$, and then place it at $(2,2,1)$ in the world space (see Figure 1.4). It is these points, position and orientations that are fed to the geometry stage.

This means that each object in the scene has its own coordinate system. Since the goal is to render to a display, i.e. represent the scene in the display's coordinate system (screen space), some transformations need to be done. These transformations are calculated and supplied by the application stage in the form of matrices that are passed on the geometry stage along with the aforementioned points, position and orientations. These points are often referred to as primitives, as they describe primitive geometric shapes such as triangles, quads or other polygons.

### 1.1.2 Geometry Stage

The geometry stage consists of several substages and are typically divided into the steps presented in Figure 1.5 [1, 14]. This stage involves lots of transformation and will be important later on in the construction of the ray marcher presented in Section 2.4.

Through the geometry stage, a model, defined as vertices (the corners of the primitives), will go from being represented in its model space to end up in screen space coordinates. The model, view, and projection transformation matrices are passed down from

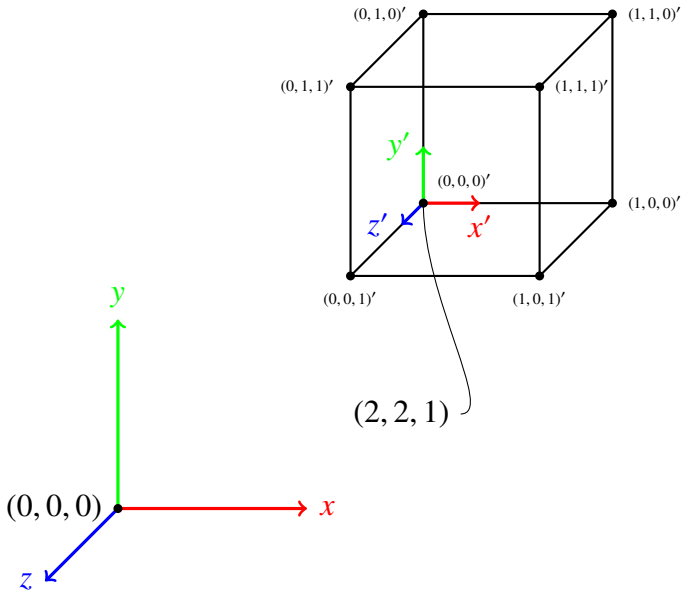



**Figure 1.4:** The model is described with its local coordinate system (model space) and is then placed in the world space by defining a position for the origin of the cube's model space. Other transformations such as rotation or scaling can also be stored for each object. Note that the coordinate systems in computer graphics are not always right-hand systems.

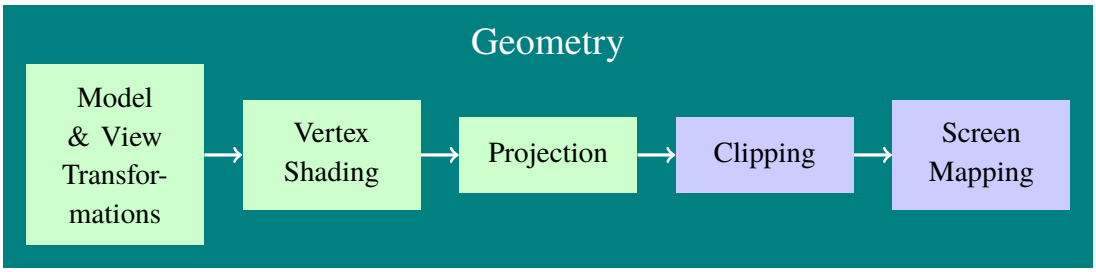



**Figure 1.5:** The substages of the geometry stage. The first three substages are part of the vertex shader and is the first programmable stage in the rendering pipeline. Source: [14].

the application stage and are typically applied in the vertex shader, which is the first programmable stage in the rendering pipeline. The vertex shader is passed the vertex attributes for each vertex; vertex attributes typically contain vertex position, vertex colour and texture coordinates. This allows us to modify the end result on a vertex basis, like calculating per-vertex lighting or applying a height map to a plane. This is known as the vertex shading. Projection is the last part of the vertex shader and allows us to for example choose perspective or orthographic projection. We will use perspective projection as this how a camera would capture a scene; an orthographic projection (also known as parallel projection) preserves parallel lines and can be used for artistic effects.

Clipping and screen mapping are the last two substages of the geometry stage. Since everything in the scene is not visible at the same time, all primitives outside of the visi-

ble scene are terminated from continuing through the pipeline and the primitives that are partially visible are clipped, or cut-off, so that the non-visible parts are removed from the pipeline. The screen mapping maps the visible scene to the coordinate system of the users screen.

### 1.1.3 Rasterisation Stage

The rasterisation stage is the stage that actually decides what colour should be assigned to a screen pixel. Just like the geometry stage, the rasterisation stage can be divided into several substages as presented in Figure 1.6.

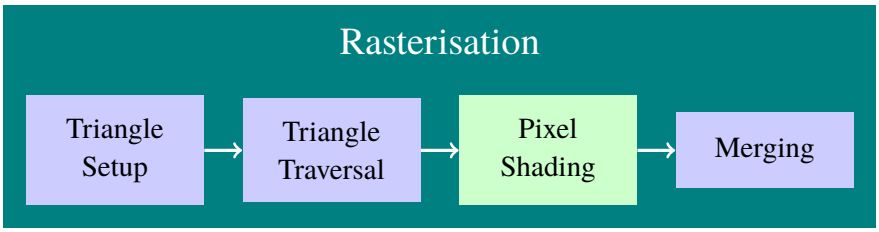



**Figure 1.6:** The substages of the rasterisation step. The pixel shading step is a programmable stage in the rendering pipeline. Source: [1].

In the triangle setup the vertex attributes of the primitives are handled and used in computation to determine what the face of the primitive should look like. The triangle traversal, sometimes called *scan conversion*, traverses the faces of the triangles to check which screen pixels they cover.

The next programmable stage of the rendering pipeline is found here in the rasterisation stage—the pixel shader[2]. The majority of the algorithms in this thesis will be performed in the pixel shader, which allows us to create effects such as per-pixel lighting and texture mapping.

The last step in the rasterisation is the merging. The colours calculated in the pixel shader are here merged with colour buffers containing prior pixel information. The merging stage also handles the pixel visibility. This is typically done using Z-buffers [1].

### 1.1.4 Display Stage

The final stage of the rendering pipeline is the display stage. This is were the results are presented on the screen. Or rather, the results are sent to a framebuffer that might be sent to the screen. We will use framebuffers to "catch" the results so that we can add additional effects before presenting them to the end user.

[2]Actually, there is the geometry shader and the tessellation shader in before, but they are optional and will not be used in this thesis.

## 1.2 Graphics Processing Units

For the rendering pipeline to work efficiently, with all its stages and transformations, the process is hardware accelerated using a *graphics processing unit* (GPU). GPUs can be integrated in the motherboard or in the CPU, or come in the form of an external graphics card. GPUs are specially built to perform graphics calculations by working in parallel, utilising vectorisation and SIMD units [1, 2].

## 1.3 OpenGL

There are different APIs to interact with the GPU. Some of the more known are DirectX, OpenGL and Vulkan. We will be using OpenGL (Open Graphics Library) to get access to the GPU. OpenGL is developed by Khronos Group and uses a extension system that allows for the latest features to be available immediately [1]. OpenGL also provides its own shader language, the OpenGL Shading Language (GLSL). It is used to write the shader programs which are the programmable stages in the rendering pipeline. GLSL is a C-like language that gives access to functions explicitly made for handling graphics. The shader programs are compiled on the GPU using the drivers provided by the chosen API.

## 1.4 Cloud Theory

Clouds are composed of small liquid droplets or crystals, mainly consisting of water, which are formed when humid air rises and expands as it reaches lower atmospheric pressure. This is because when the air expands, the temperature falls, and the water vapour in the humid air condenses. The formation and structure of clouds are very dynamic and are mainly the result of vertical motions. We will consider cumulus clouds, which get their vertical motion from convection, for our implementation. These vertical motions can be compared to a dense and coloured liquid being dropped into a water tank, creating a circulating vortex [10]. The convection plays a big part in the features of the clouds. Another aspect that contribute to the features of the clouds is the droplet size distribution. The droplet size distribution affects the Mie scattering, which accurately describes how the photons scatter in water droplets for different angle of approaches [3]. Figure 1.7 shows the Mie scattering phase function as calculated by Philip Laven's MiePlot software [9]. This software allows for a variety of different situations to be constructed by entering different values for air humidity and temperature, mean droplet size, droplet size distribution, the wavelength of the light, etc. We will not consider different situations or wavelength dependent scattering but rather compare this performance intensive scattering to another, computationally cheaper, type of scattering. In this case an arbitrary wavelength in the visible spectrum is chosen as well as normal temperature and pressure for the atmospheric settings. As mentioned earlier the droplet size distribution affects the scattering and is chosen as a modified Gamma distribution as discussed by Bouthors et al. [3] and Mason [10]. The plot in Figure 1.7 shows that the majority of light is scattered forward, however there are some other peaks as well resulting in some interesting phenomena. The peak around 140° is the phenomenon known as a fog bow and the peak at 180° is known as a

glory. See Figure 1.8 for a photograph of these phenomena. The larger circle is the fog bow and the smaller circle in the middle, surround the shadow of the photographer, is the glory.

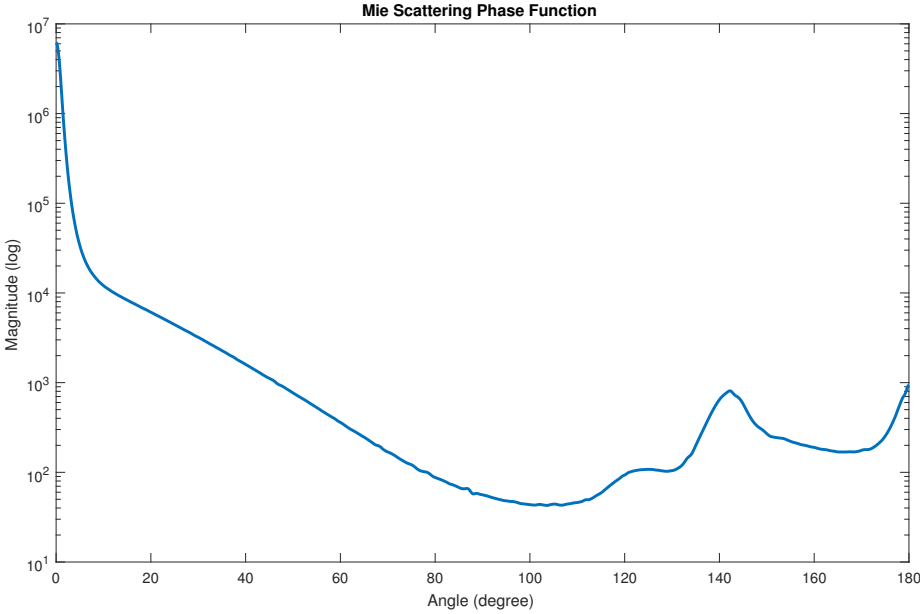



**Figure 1.7:** A plot showing the intensity of the light as it scatters in different directions.

**Figure 1.8:** A photograph showing the fog bow and the glory. Source: [7]

## 1.5 Related Work

Bouthors et al. [3, 4] discuss the usage of Mie theory for calculating the light scattering in clouds. They also resort to Philip Laven's MiePlot software to attain realistic light

scattering functionality, but have a different approach for the higher orders of the light scattering and use collector slabs on the surface of the clouds to calculate the incident light.

Harris and Lastra [6] use a implementation similar to our but handle their multiple forward scattering using bi-directional scattering distribution functions (BSDF). For the phase function, they use Rayleigh scattering and mention that this could be substituted for a more physically based function.

Schneider and Vos [13] also present a implementation very similar to our but do not discuss the usage of a more physically based phase function, like the Mie phase function.

## 1.6 Contributions

Our implementation compares the usage of a simpler phase function (Henyey-Greenstein) to a more physically based one (Mie) and what this means practically, as the Mie phase function is to complex to calculate in real-time. We also present a way to construct clouds by carving cloud shapes out from blocks of noise textures.

# Chapter 2
# Implementation

In this chapter we will discuss the different techniques explored to find desirable performance while not compromising the visual results too much. The first section presents the outline of the cloud generation and rendering before going into implementation details.

## 2.1 The Cloud Rendering Algorithm

To render clouds realistically, we would need to trace each photon radiating from the sun, follow their paths as they scatter through the clouds and register those few photons that end up in the camera. This is of course computationally very intensive and we do not want to trace all photons that never end up in the viewer's perspective. This is why our cloud rendering algorithm is built upon a ray marching technique, which means that we reversely trace a ray from the camera into the scene instead. The rays that are cast from the camera perform equidistant samples to determine whether a cloud is to be rendered or not. The clouds are procedurally generated and stored in 3D textures that are placed in the scene. A second ray marching is performed at each sample point inside a cloud. This ray is cast toward the sun to perform lighting calculations. This means that we are only considering the first order of scattering. In Section 2.5 an approximation for the higher order of scattering is presented. Figure 2.1 shows a basic setup of the whole cloud rendering algorithm.

## 2.2 Noise Functions

The typical way of rendering procedural clouds—or anything procedural for that matter—is to use noise functions, whether it be 2D clouds or volumetric 3D clouds. A completely white noise lacks structure and is not too pleasant to look at, as it does not resemble anything, so in computer graphics there are different ways to modify the white noise and create

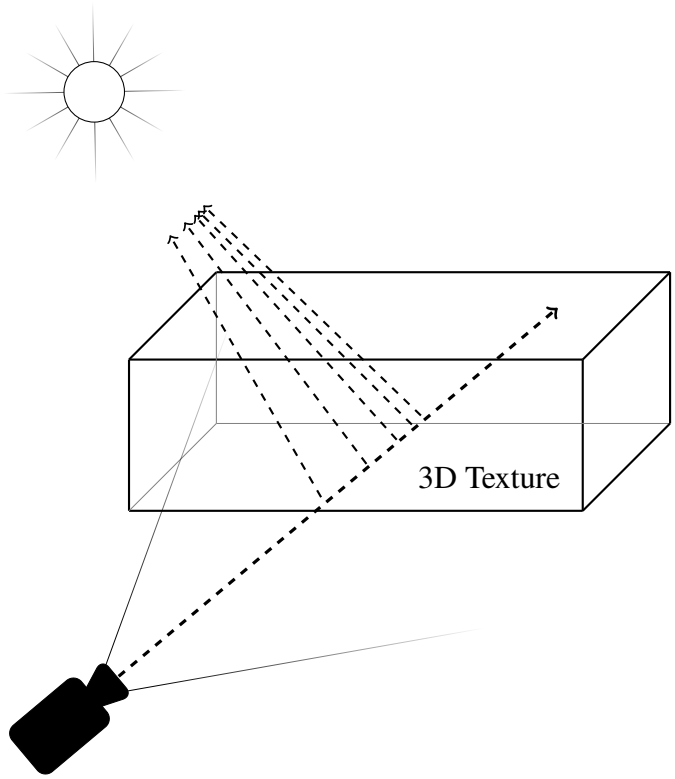



**Figure 2.1:** A ray is cast from the camera for every pixel on the screen. The ray marches through the scene, sampling at pre-set intervals. If the ray samples a point inside of a cloud, a second ray is cast towards the sun to calculate how much light reaches that point.

something with a bit more structure without introducing artefacts. This usually means that we want to keep the gradient continuous for the produced texture, i.e. the texture should be in $C^2$. We also want to construct cloud covers larger than the texture, without having visible edges, so the textures need to wrap continuously over the edges of the texture.

The noise generation algorithms presented below can be quite costly and the resulting textures are therefore precalculated and stored on the hard drive.

### 2.2.1 Perlin Noise

In 1983 Ken Perlin developed a gradient noise, now commonly known as *Perlin noise*. Perlin has since improved his algorithm [12, 11] into what sometimes is referred to as *improved Perlin noise*. It is this improved version that we use.

Perlin noise is lattice based and is generated by assigning a random gradient vector to each intersection of the lattice (see Figure 2.2). When the value of the noise is read at a specific coordinate, the lattice cell wherein the coordinate lies is determined and the value is calculated by interpolating the cell's corner gradients using dot product and linear interpolation. Figure 2.3 shows what the resulting noise texture might look like. By assigning the gradient vectors to the lattice in a repeating pattern, a wrapping texture can be attained.

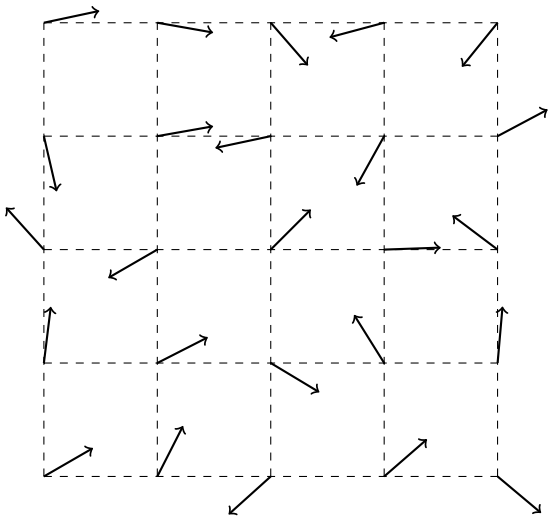


**Figure 2.2:** Perlin noise uses a lattice with randomly assigned unit vectors that denote the gradients of the texture at the lattice intersections.

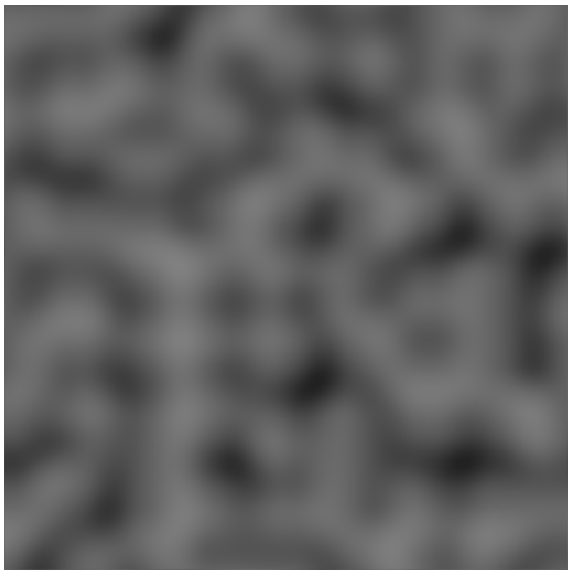


**Figure 2.3:** This is what the Perlin noise looks like. The texture seems random, yet it has some structure making it suitable as a base texture when mimicking natural phenomenons, like wood and marble surfaces or clouds. We use 3D textures and this image shows a cross-section of a Perlin noise 3D texture.

### 2.2.2 Cellular Noise

Cellular noise, also commonly called Worley noise or Voronoi noise, is a point based noise as opposed to the lattice based Perlin noise. It was first introduced by Steven Worley in 1996 [16]. The idea is to take random feature points in space and then for every point in the space assign the value which corresponds to the range to the closest feature point. In our 3D implementation of cellular noise, we divide the space into cells and assign one

feature point per cell. This means that the closest feature point to a sample point must be in either the same cell or one of the neighbouring cells as shown in Figure 2.4. Just as with the gradient vectors in the Perlin noise, the feature points are generated in a repeating pattern, with one iteration of the repetition exactly fitted into the texture to make sure that the texture wraps continuously at the edges.

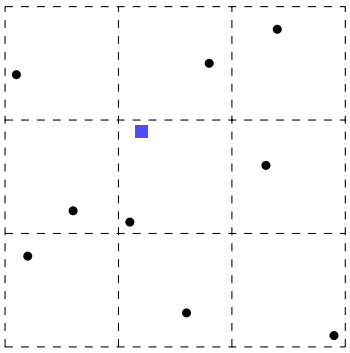


**Figure 2.4:** This is what the cellular noise setup looks like. The black circles are the randomly chosen feature points. In this setup exactly one feature point is present in each cell. When the noise function is sampled (at the blue square), only the immediate neighbouring cells need to be checked for the closest feature point. This means that a $3 \times 3 \times 3$ block of cells needs to be analysed for each sample point when using a 3D texture.

To get a result that works well for cloud generation, we do not sample the shortest distance to a feature point, but rather the theoretical maximum distance minus the shortest sampled range. For cells consisting of unit cubes, the theoretical maximum distance (the space diagonal) is $\sqrt{3}$. The values for the texture are hence calculated as (in this function the value gets normalised as well to get a result in the range [0,1])

$$value = 1.0 - \frac{shortest\_distance}{\sqrt{3}}.$$

This gives a result with ball-like features around each feature point which better imitates the look of a cloud. A cross-section of our 3D cellular noise is presented in Figure 2.5.

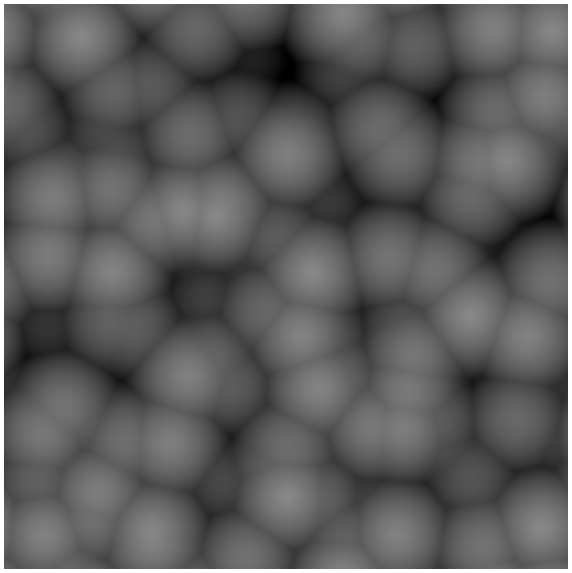
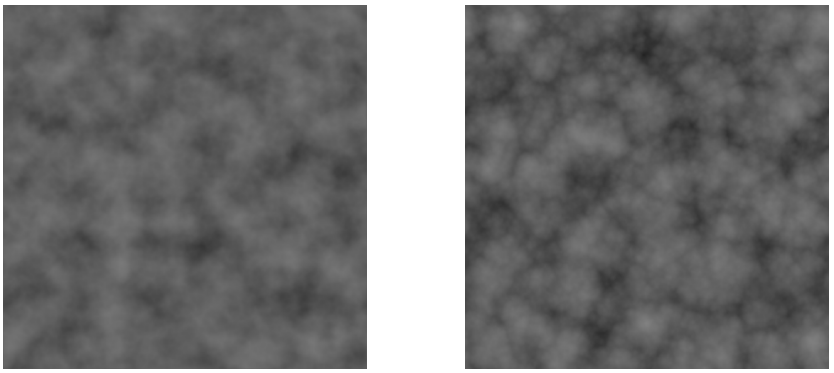


**Figure 2.5:** This is what cellular noise looks like. We can see the ball-like features.

### 2.2.3 Fractional Brownian Motion

The noise texture described above forms the foundation for our cloud textures. The next step of our noise generation is called *fractional Brownian motion* [17], often abbreviated as fBm, which when applied to noise textures renders results that look like smoke or clouds. The fBm noise is constructed by layering a noise texture of different sampling frequencies on top of itself. By changing the frequency in powers of 2 the wrapping effect can be preserved. The number of layers is usually called octaves and Figure 2.6 shows the 5 octave fBm version of our textures.



**(a)** **(b)**

**Figure 2.6:** **(a)** This is Perlin noise with 5 octaves of fBm. Compared to Figure 2.3, we can see that the underlying structure is the same but the overall noise texture has much more finer details. **(b)** This is the cellular noise with 5 octaves of fBm. This texture has more distinct structure with deeper cracks and resembles the texture of a cauliflower.

## 2.3 Cloud Forming

To get the actual cloud shapes from the noise textures, several step are conducted to achieve the different features of clouds, with these features being the flatter bottoms of clouds and the towering tops that are the results of the convection mentioned in Section 1.4.

### 2.3.1 Height Distribution

We have a height distribution function in our implementation which governs the shape of the clouds to be denser toward the bottom of the 3D texture and gradually thinner towards the top. The function looks like this:

$$\mathrm{HD}(h) = (1 - e^{-50 \cdot h}) \cdot e^{-4 \cdot h},$$

where $h$ is the height. This gives the clouds a high probability of existing near the bottom of the texture, effectively giving the clouds a flat bottom, as well as retaining a slight chance of towering clouds. The choice of using a function of the form $f(x) = (1 - e^{-a \cdot x}) \cdot e^{-b \cdot x}$ is based on that it gives an easily accessible way of tuning the result, by just altering $a$ and $b$. $a = 50$ and $b = 4$ were found by trail and error and dictates the sharpness of the falloff in the bottom and top respectively. Figure 2.7 shows the function applied to the noise based 3D texture.

**Figure 2.7:** The height distribution function is applied uniformly for the whole 3D texture by simply multiplying, for each point of the texture, the texture with the output of the function; the height coordinate of the point in the texture is fed to the HD-function. This does not disrupt the edges were the texture wrap.

### 2.3.2 Gaussian Towers

Since the height distribution function is applied uniformly, no distinct cloud tower will emerge. An additional height function in our implementation allows for creation of peaks or towers in the 3D texture. We choose to implement the towers as multivariate normal distribution peaks. This gives us an easy way of moving and modifying the shape of the peaks. The only limitation with this approach is that the peaks may not be placed too close to the edges so that they disturb the continuity of the texture wrapping.

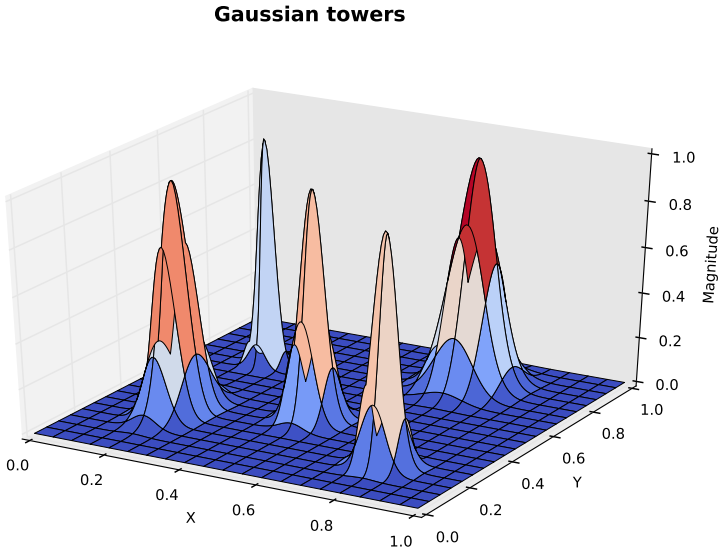



**Figure 2.8:** This is an example of what the Gaussian towers look like. Note that the function is zero along the edges so that when this function is added as an offset value to the 3D texture, the texture wrapping is kept intact.

### 2.3.3 Finalisation

The last step of the cloud forming was to use a threshold function—a simple cut-off value. This cuts away the "thinner" parts of the 3D texture and shows more distinct clouds. Figure 2.9 shows a cross-section of the final 3D texture.

The 3D texture used in our implementation has noise of different frequencies in the four colour channels of the texture. This means that with a single texture read we can retrieve the shape of the clouds from one channel, as well as other noise functions at higher frequencies. This allows us to apply finer detail and easily tune the end result by mixing in different amounts of each channel in the base cloud shape.

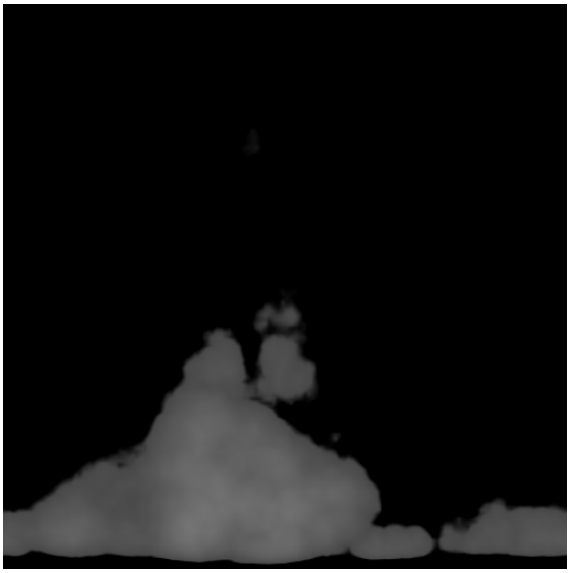


**Figure 2.9:** This is what the 3D texture looks like after a threshold function is applied.

## 2.4 Volumetric Ray Marching

The details of the implementation discussed until now were computed offline (not in real-time) and saved to textures. The ray marching is done in real-time on the GPU so the implementation must take performance hits into consideration.

### 2.4.1 Deferred Shading and Ray Casting

To be able to perform the ray casting, deferred shading is used. Deferred shading means that the scene is first rendered without the clouds and then, deferred to a later stage, the clouds are rendered. The whole scene, excluding the clouds, goes through the rendering pipeline and is rendered to a texture that is assigned the framebuffer. A second render pass is then performed by rendering a primitive which covers the entire screen, i.e. a full-screen quad. This allows us to access each pixel on the screen in the pixel shader. To cast the rays into the scene, the screen coordinates need to be transformed into world space coordinates. This involves reversing all the transformations discussed in Section 1.1. As the ray is cast through the scene, it samples at equidistant point to determine whether a cloud is to be rendered or not, as discussed in Section 2.1. The first time cloud is encountered by the ray marcher, the 3D texture's value at that sample point is stored as an alpha value [1]. Subsequent non-zero samples are added to this value. When the alpha reaches a value of 1, the alpha value is saturated and we know that none of the colours from the scene texture will show through. The ray marching is then halted. An upper limit to the length of the ray is set so that it eventually stops even if it never encounters any clouds. After the ray marcher stops, the colour of the clouds (this is calculated as described in Section 2.5) is blended with the pre-rendered scene texture using the sampled alpha value. See Listing

[1] An alpha value is used to describe the translucency of pixel.

2.1 for a clearer overview of the ray marcher.

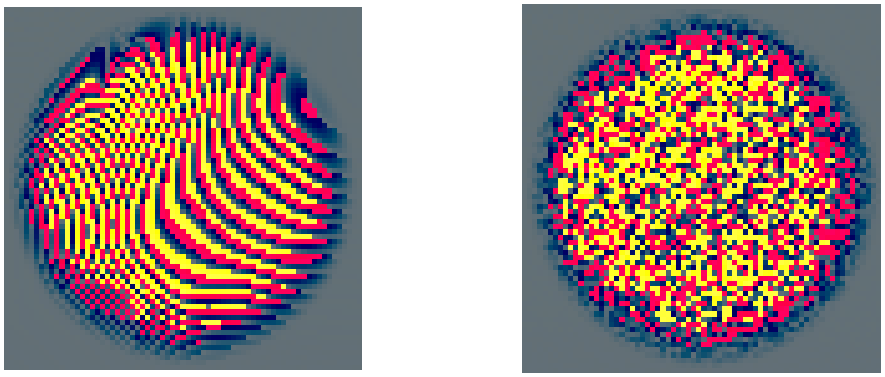
```
1  vec4 cast_ray(vec3 origin, vec3 dir) {
2
3      vec4 value = vec4(0.0);
4
5      float delta = 1.0;
6      float start = gl_DepthRange.near;
7      float end = 500.0;
8
9      for (float t = start; t < end; t += delta) {
10
11         /* Calculate new sample point */
12         sample_point = origin + dir * t;
13
14         /* Stop rays that already reached full opacity */
15         if (value.a == 1.0) {
16             break;
17         }
18
19         value.a += cloud_sampling(sample_point, delta);
20         value.a = clamp(value.a, 0.0, 1.0);
21
22         /* Calculate cloud colours from shadows and scattering */
23         ...
24         value.rgb += ... * alpha;
25         }
26     }
27     return value;
28 }
```

**Listing 2.1:** This listing shows a heavily condensed version of the ray marcher implemented in GLSL. `cast_ray()` casts a ray from `origin` in the direction of `dir`. Here the step length is called `delta` and is set to 1 unit. The upper limit for the ray marcher is set to 500 units. On line 23 a second ray marcher is placed to calculate the lighting effects.

### 2.4.2 Step Length

The ray marchers performance impact is highly dependent on the amount of samples (or steps) it needs to take for each pixel. The amount of steps can be reduced by increasing the distance between the steps. If the distance is increased too much, artefacts—in the form of colour banding—will start to appear. There are ways to reduce the banding effect [5] but a very simple approach is to add some small random length to the first step of the march. This distorts the structure of the banding and it becomes less visible. In Figure 2.10 some artificial banding effects have been constructed to show what the result of using a random step length at the start of the ray marching looks like.

An adaptive step length is an approach tested in our implementation to try to reduce the amount of steps taken in the ray marcher. This means that after some distance we allow the steps to increase in length. This means that a lot of computational power can be saved when rendering clouds far away. But for saving computational time for clouds close to the viewer, a second approach to this is also implemented and discussed next.



**(a)** **(b)**

**Figure 2.10:** **(a)** The curved lines are a result from a too long step length. Typically the banding would not be this severe; this is an extreme example of banding. **(b)** After applying some random step length to the first step, the banding issue is less obvious. This method works best for banding with high frequency. If there had just been a couple of bands across the circle, they would still be visible after applying the noise, although their edges would be less defined.

## Preprocessed Cloud Structure

We want to exploit the fact that we already know exactly where the clouds are in the 3D texture. At program start, before the rendering starts, we put a preprocessing step of the cloud texture. In this step a low resolution structure is constructed such that it completely encloses the clouds and is stored in another 3D texture. Figure 2.11 shows cross-sections from these 3D textures.

The low resolution structure is used as a boundary to determine whether small or large step lengths should be used. The ray marcher is altered to accommodate for the low resolution structure:

- The ray marcher starts with a larger step length. The larger step length is set so that it corresponds to less than a pixel length in the low resolution structure.
- When the ray marcher starts to sample non-zero bits from the low resolution structure, it takes one large step backward and changes to a small step length. The initial step backward is necessary so that no “cloud bits” are missed.
- The ray marcher continues with the smaller step lengths and samples the cloud texture instead. Every time the small steps adds up to a large step length, the low resolution structure is sampled to determine if it can change back to the large step length or not.

Going back to Figure 2.11, we can see that the structure adds a broad margin to the actual cloud. The preprocessing step is implemented to add double padding to the clouds so that the ray marcher is not able to cut the corners of the structure, which would result in ugly artefacts. This is because the structure is constructed so that the larger step length is

**(a)** **(b)**

**Figure 2.11: (a)** This is the 3D texture that is passed through the preprocessing stage. **(b)** This is what the low resolution structure looks like. In this figure it is stretched to the same size as the cloud texture to demonstrate that it encapsulates all "cloud bits" in the original texture.

just a bit shorter that the side of one structure cell. The ray marcher will detect the cloud structure when approaching head-on but in the cases where the observer is not aligned with the grid pattern of the cloud structure, the ray marcher may take a step past the corner of the cell without detecting it. This is why double padding was used, and means that a second layer of cloud structure encloses the actual clouds even if the outermost layer does not contain any cloud to be rendered. It is only there to make sure that we are trying to sample the high resolution cloud texture when we are in the proximity of it.

## 2.5 Lighting Calculations

There are two types of light effects considered in our implementation. The first one is the attenuation of the light as it passes through the cloud. This effect can be calculated using Beer-Lambert's law

$$\mathrm{T}(d) = e^{-m \cdot d},$$

where T is the transmitted light, $m$ is a material dependent variable and $d$ is the length the light travels through the material [15].

The second effect considered is light scattering. When the light from the sun reaches the water droplets in the clouds, it scatters according to Mie theory as discussed in Section 1.4. To calculate how all these light rays scatter indefinite times in the cloud in real-time is not feasible, so the scattering is split up into two parts. (There are other ways to handle the different orders of scattering [3, 4], but this is how we decided to do it in our implementation.) One part deals with proper angle-dependent scattering as shown in Figure 1.7. This is only done for a single light scatter. The the other part is an approximation of multiple scattering within the cloud. The following function is our approximation of the

in-scattering in the clouds:

$$S(d) = 1 - e^{-c \cdot d},$$

where S is the intensity of the scattered light, $c$ is a variable that determines how fast the scattering effect builds up in the cloud, and $d$ is the length the light travels through the cloud. This function together with Beer-Lambert's law constitutes our light function (see Figure 2.12). This way of calculating the light was discussed by Andrew Schneider of Guerrilla Games at SIGGRAPH 2015 [13].

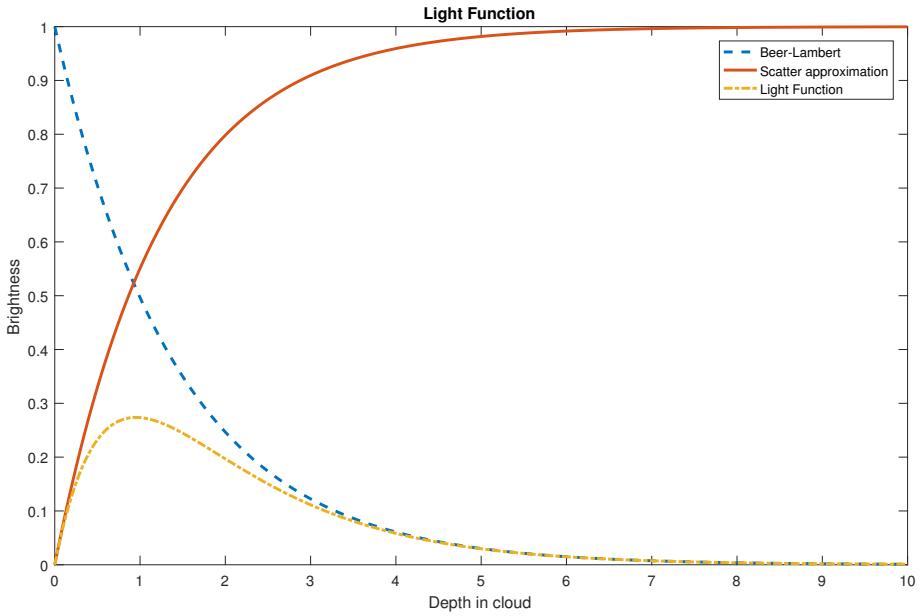



**Figure 2.12:** The plot shows Beer-Lambert's law and our scatter approximation function. Multiplied together they create our light function. We can see how a light ray that passes through a cloud first starts to scatter but as the cloud attenuates the light, the brightness falls off.

The clouds are given a dark base colour at the start of the ray marching, to which a brighter colour is added depending on how exposed to the sun they are. As mentioned earlier, the lighting calculations are implemented using a ray marcher as well. Every time the cloud ray marcher samples a point inside a cloud, the ray marcher for the light calculations samples the cloud texture towards the sun to determine how deeply embedded in the cloud the original sample point is. This depth is passed to the light function which calculates how much bright colour should be added to the dark base colour.

## 2.6 Phase Function

The Mie scattering phase function in Figure 1.7 can be used for the single light scatter by saving the values from Philip Laven's MiePlot software to a texture. This texture can then be used as a look-up table for different angles, which are the angles between the sample point and the sun as seen by the camera.

A common way of approximating the Mie scattering phase function is to use the much simpler Henyey-Greenstein phase function [4]. The Henyey-Greenstein phase function is given as

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

where HG is magnitude of the scattered light, for a certain angle $\theta$, and $g \in [-1, 1]$ is a parameter which determines the concentration of the scattering; a negative $g$ gives backward scattering [8]. This function is plotted in Figure 2.13.

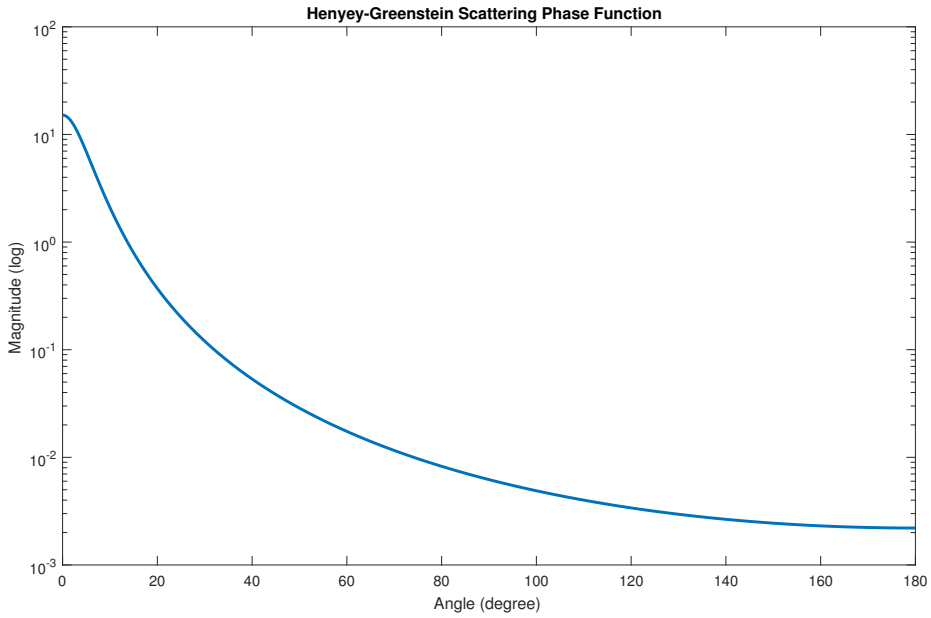



**Figure 2.13:** This is is the Henyey-Greenstein scattering phase function for $g = 0.9$. This gives a high concentration of forward scattering much like in Figure 1.7 but it lacks the peaks around 140° and 180°.

It is possible to implement the Henyey-Greenstein phase function directly in the fragment shader as opposed to the much more complicated Mie phase function. This saves the one texture read that is involved when using the Mie phase function. In our implementation we try both using the precomputed Mie phase function and the fragment shader implemented Henyey-Greenstein phase function.

## 2.7 High Dynamic Range Lighting

Usually the colour channels in the framebuffers are stored as 8 bit values, ranging from 0 to 1. This means that the colour channels can only assume 256 different values. This works for most applications, but the nature of the clouds and sun gives us very bright areas on the screen. This can lead to parts of the screen losing important colour information as the colours get rounded to the nearest of the 256 states. A way around this problem is to use high dynamic range (HDR) lighting. Instead of binding textures with 8 bit values for each colour channel to the framebuffers, a floating point value is used. In our implementation we use a 32 bit floating point value for each channel. HDR lighting also allows us to assign colour values above 1 to the framebuffer. Before the final framebuffer is rendered to the screen, the values need to be transformed back to the range 0 to 1. This is done using what is known as tone mapping. This is the tone mapping used (from [1]):

$$L_d(x, y) = \frac{L(x, y)}{1 + L(x, y)},$$

where $L(x, y)$ is the luminance of the pixel at $(x, y)$ and $L_d(x, y)$ is the tone mapped value in the range 0 to 1. This tone mapping compresses high luminance values while retaining more information for the low luminance values [1].

The HDR lighting also gives the ability to implement some extra effects. Light blooming is such an effect. It allows for bright areas to bleed in to neighbouring areas which results in some smoother light transitions around the bright areas and gives the bright area a glowing feature. This is implemented by extracting the bright pixels, above some threshold, and blurring these using Gaussian blur. Theses blurred pixels are then added on to the scene texture before the tone mapping.

# Chapter 3
# Results

## 3.1 Hardware and Software

All results are captured on a Windows 10 machine with a Intel Xeon E5-1620 v3 running at 3.50 GHz and 64 GB of RAM. Even though the CPU and RAM are high end, the GPU is provided by an Nvidia GeForce GTX 680 card which can be considered a mid-tier card at the time of writing. The results are captured at $1280 \times 720$ resolution.

The OpenGL context is provided by Simple DirectMedia Layer (SDL) and OpenGL Extension Library Wrangler (GLEW) is used to provide OpenGL extensions. Microsoft Visual Studio 2013 is used to compile the C++ code. The complete implementation can be found here (ray marcher):

- `https://github.com/rikardolajos/clouds`

and here (texture generation using noise):

- `https://github.com/rikardolajos/noisegen`.

A benchmarking test track was constructed for the camera to allow for fair comparisons between the different implementations. In this chapter we will compare the frame times along the test track for the different algorithms to better understand their performance hit. The test track is constructed so that is passes through different situation in the scene (e.g. below clouds, above clouds, inside clouds, etc.) so that we can see which algorithm are preferable in which conditions. For all plots showing frame time in this chapter, less is better.

## 3.2 Collected Data

This section presents frame times collected along the test track. Figure 3.1 and Table 3.1 show the result using the preprocessed cloud structure. While utilising the preprocessed cloud structure, we can see an overall improvement in frame time except for the part where the ray marcher gets saturated early—when the camera is inside the clouds.

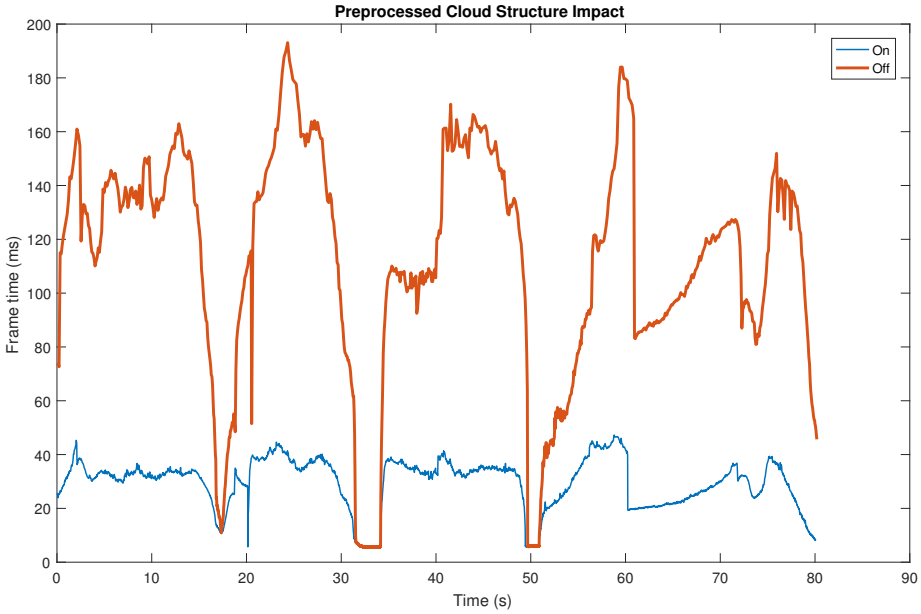



**Figure 3.1:** A plot showing the performance when using the preprocessed clouds structure compared to not using it. The sections where the graphs overlap is when the camera passes through the clouds.

| Preprocessed cloud structure | Average frame time | Average frequency |
|---|---|---|
| On | 23.728955 ms | 42.142606 FPS |
| Off | 56.155690 ms | 17.807635 FPS |

**Table 3.1:** The preprocessed cloud structure performs on average about 32 ms faster.

In Figure 3.2 and 3.3 the result of using an adaptive step length is shown. The average frame time and frame render frequency is presented in Table 3.2. In general the adaptive step length does not have a noticeable impact on the frame time. It only affects the parts where the camera is placed far from the clouds (see around 60 s in Figure 3.2). For these measures large step length was set to 20 units while the adaptive step length started at 1 unit, and for samples more than 100 units away they increase with 1 unit each unit.

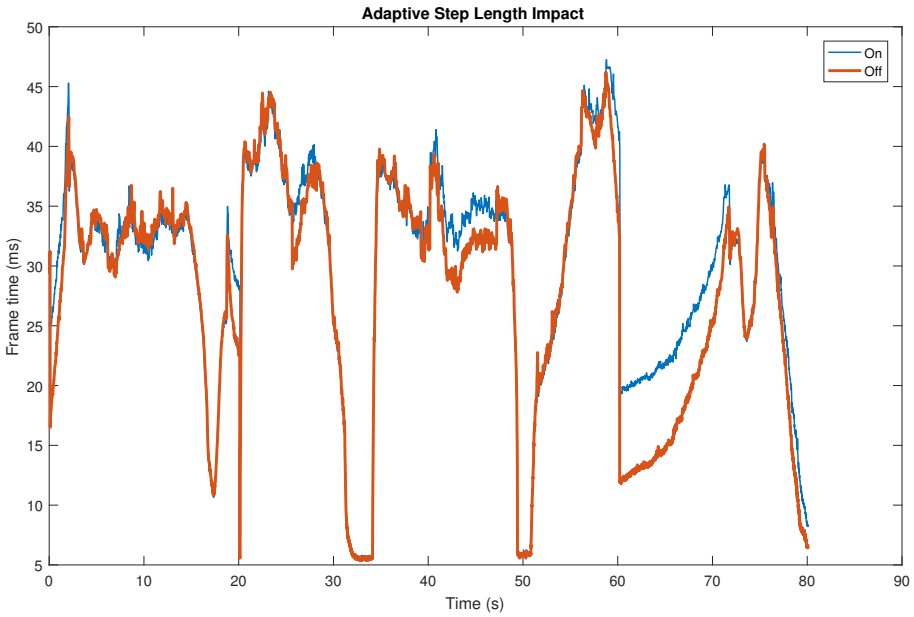



**Figure 3.2:** This plot shows an adaptive step length compared to a static step length for the smaller steps. Around the 60 s mark the camera is far away from the clouds and the graphs are showing very different results.

| Adaptive step length | Average frame time | Average frequency |
|---|---|---|
| On | 21.761936 ms | 45.951795 FPS |
| Off | 23.728955 ms | 42.142606 FPS |

**Table 3.2:** On average the saved frame time is about 2 ms when using the adaptive step length.

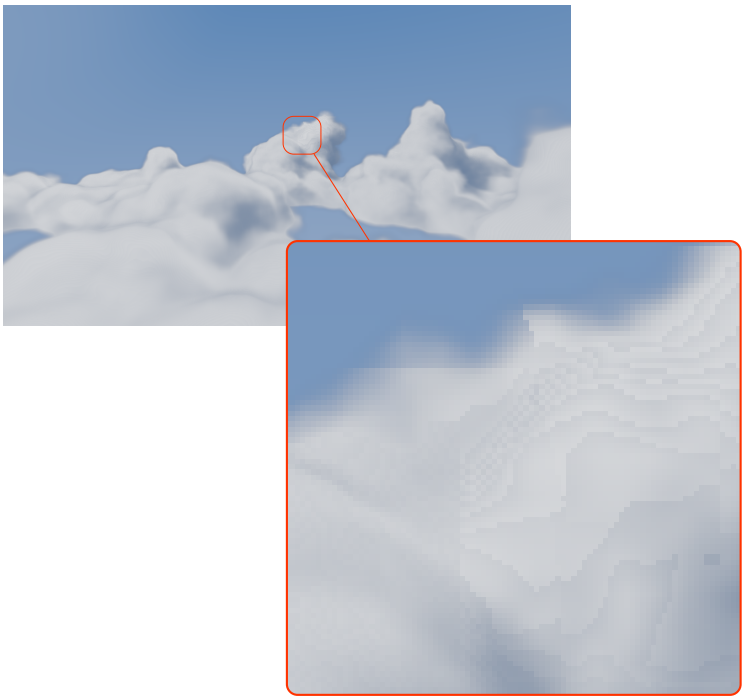


**Figure 3.3:** This figure shows a zoomed in part of the clouds when using an adaptive step length. Far from the camera, where the adaptive steps are allowed to grow large, some artefacts start to form.

Figure 3.4 and Table 3.3 present the impact the overall lighting calculations have on the rendering, i.e. the impact of running the ray marcher for the lighting. We can see that the light calculations affect the frame time regardless of the camera position.

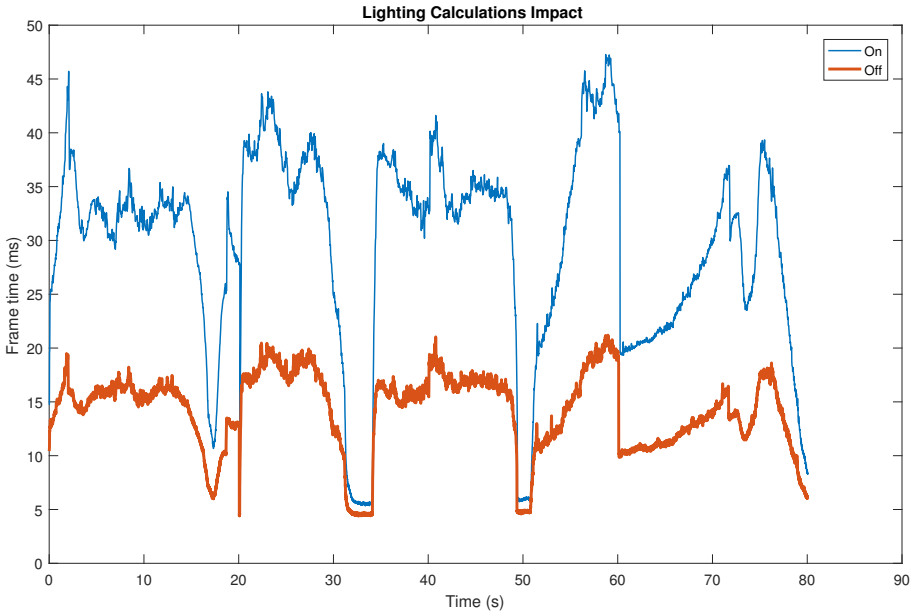



**Figure 3.4:** The green line is for the clouds rendered without using light calculations. The blue line shows the frame time when light calculations are performed.

| Light calculations | Average frame time | Average frequency |
|---|---|---|
| On | 23.764996 ms | 42.078694 FPS |
| Off | 12.553803 ms | 79.657134 FPS |

**Table 3.3:** The ray marcher for the lighting calculating takes about 11 ms on average during the test.

The impact of reading the phase function from a precomputed texture compared to calculating it on-the-fly in the fragment shader is presented in Figure 3.5 and Table 3.4. We can see that the texture reading takes longer time to perform than computing the phase function in the fragment shader.

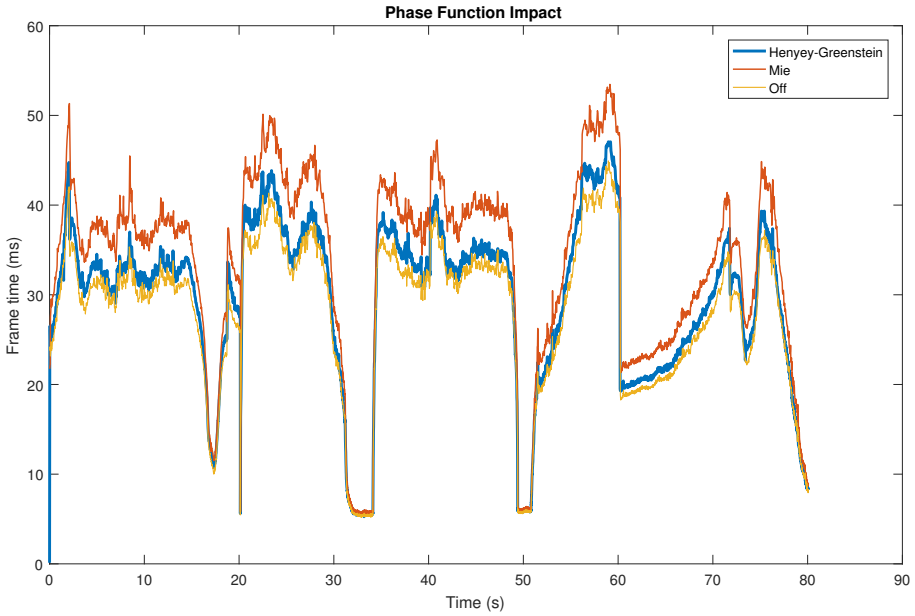



**Figure 3.5:** The Henyey-Greenstein phase function (calculated on the fragment shader) compared to the Mie phase function (read from a texture). The third graph shows the frame time when no phase function is used.

| Phase function | Average frame time | Average frequency |
|---|---|---|
| Henyey-Greenstein | 23.643466 ms | 42.294982 FPS |
| Mie | 25.999220 ms | 38.462692 FPS |
| Off | 22.585036 ms | 44.277105 FPS |

**Table 3.4:** Calculating the phase function on the fragment shader (Henyey-Greenstein) takes about 1 ms while reading from a texture (Mie) takes a little over 3 ms.

## 3.3 Rendered Images

This section presents some rendered images (Figures 3.6 through 3.9) of the final implementation. These images were constructed using the preprocessed cloud structure, a static step length and the Henyey-Greenstein phase function. The noise textures are completely based on the cellular noise. The images in Figures 3.10 and 3.11 shows the visual difference between using the Henyey-Greenstein versus using the Mie phase function.

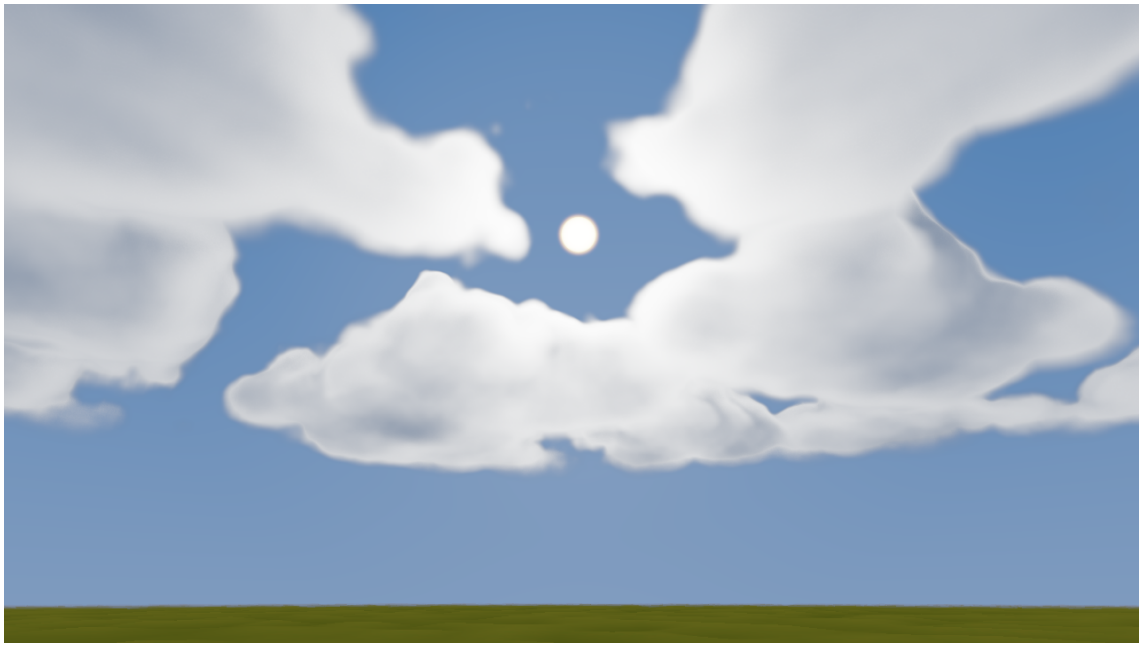


**Figure 3.6:** In this image the camera is placed on the ground and looking up toward the sun.

**Figure 3.7:** The sun is off to the left in this image. The camera is placed up amongst the clouds and the shadows, as a result of Beer-Lambert's law, are clearly visible.

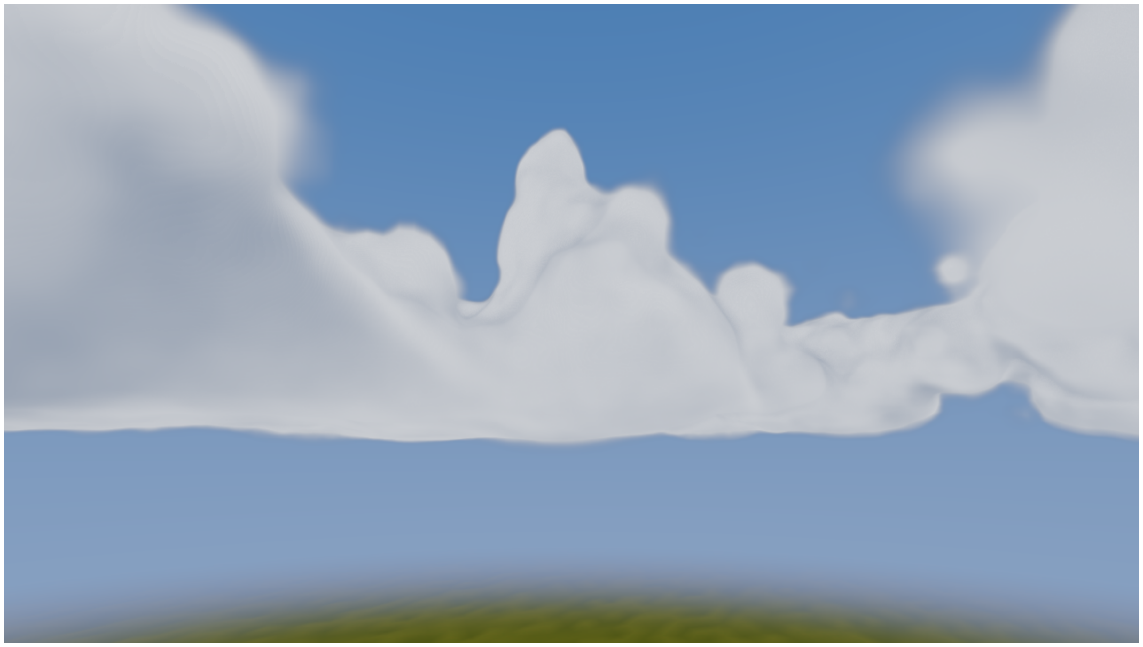


**Figure 3.8:** The camera is placed so that we can see the flat bottoms of the clouds. The sun is behind the camera. The darker contours of the cloud in the centre of the image (in front of the towering cloud) are a result of the "faked" in-scattering effect from the light function.

**Figure 3.9:** Here the camera is again placed on the ground and looking up towards the sun. The effect of the phase function can be seen here as the part of the cloud that is covering the sun is almost glowing.

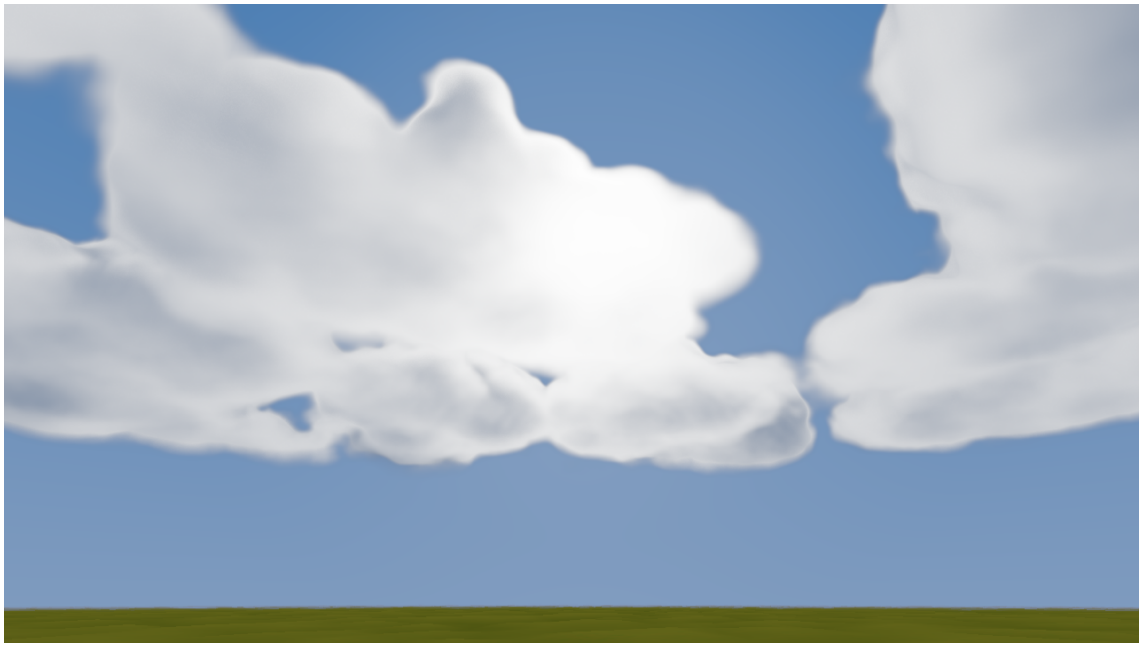


**Figure 3.10:** This is an image showing the Henyey-Greenstein phase function.

**Figure 3.11:** This is an image showing the Mie phase function.

## 3.4 Rendered Video

A video is also available. This gives the *real-time* aspect better justice than the images in Section 3.3. The video is available here:

- `https://rikardolajos.se/archive/clouds/video.php`
- `https://www.youtube.com/watch?v=V-Ij1HPSkb8` (mirror)

The first part of the video shows the test track used for the data collection. After the cut we try to better show the effect of the phase function. Same implementation settings as used for the images in Section 3.3 are used for the video.

# Chapter 4
# Discussion

The two main parts of this thesis were the forming of the clouds and rendering the clouds in actual real-time without losing too much of the visual details. The cloud forming is harder to evaluate objectively, while the cloud rendering has the frame times from Section 3.2 as concrete evaluation.

## 4.1 Cloud Forming

In our implementation both Perlin noise and cellular noise were tested. We test using plain Perlin noise, plain cellular noise and a mix of both as the base for our 3D texture. The differences in the final result are not very distinct and are hard to present in the result chapter. The images presented in Section 3.3 are based completely on the cellular noise since the cellular noise gives rounder structures to the clouds. This is partly because clouds are not formed completely randomly, like Perlin noise which is based on the randomly assigned gradient vectors, but rather have a under lying structure as result of the convection.

The clouds in Figures 3.6 through 3.9 lack some higher resolution details. Our implementation makes use of higher frequency noise, placed in different colour channels of the 3D texture, which is visible in the shaded part of Figure 3.7. But the overall surface does not have the finer details of real clouds as shown in the photo in Figure 1.1.

A way of obtaining finer details is to use a higher resolution for the 3D texture; we use a $128 \times 128 \times 128$ resolution for the texture. But when raising the resolution, the following points needs to be considered:

- if Perlin noise is used, Perlin's implementation of the noise must be altered if we want resolution over $256 \times 256 \times 256$ as it natively does not support this,
- the construction of the 3D-texture takes $O(n^3)$, so just a doubling in resolution takes 8 times longer in construction,

- and if we use the preprocessing of the cloud structure, this also takes $O(n^3)$.

If the construction of the 3D texture takes a longer time, this might not pose a too large implication, as the 3D texture is saved and read from the disk, but the preprocessing and construction of the low resolution structure is performed at program start and each time the clouds change. This would lead to heavily increased loading times in an end-product.

## 4.2 Cloud Rendering

Lowering the amount of steps that are taken in the ray marcher is essential for keeping the frame times low. Saving on average 32 ms in rendering time shows that the preprocessed cloud structure works well in the test track case. A great advantage of this method for reducing the number of steps is that it does not affect the rendered result, because the smaller steps are always used while inside the clouds. The drawbacks of using this low resolution texture is that the preprocessing takes time and that we must know where the clouds will be rendered. If we want to generate the clouds in the fragment shader, the low resolution structure would not work without reconstruction. Generating the clouds like this would allow for animation to be implemented easily as the clouds would not be based on the static 3D texture. Another way of implementing animation of the clouds would be to use 4D textures. This could give us the possibility to also use a 4D texture for the low resolution texture, and again probably save a lot of rendering time.

The adaptive step length is another way of reducing the number of steps taken. This implementation is more of a balancing act as noticeable improvements in frame times (about 2 ms) does not occur until severe banding and artefacts starts appearing in the distant clouds (see Figure 3.3). In addition, the gained performance is dependent on where the camera is placed in relation to the clouds.

The light calculation takes a lot of time to perform. The 11 ms it takes could be valuable in an end-product, where the allotted 33 ms can not all be spent on rendering just the clouds in the scene. In such cases the ray marcher for the light calculation could be completely scrapped in favour for some more “faked” and less computationally expensive lighting calculations.

It is interesting to see that the phase function that is read from a texture takes longer to render than the one that is computed on-the-fly; the Mie phase function is more than three times slower than the Henyey-Greenstein phase function. This is the result of memory bandwidth. Reading from texture result in memory bandwidth being used and can be quite expensive. The result of using this kind of angle dependent scattering for only the first order of scattering gives an acceptable visual result and the effects from the phase function that are present in Figure 3.9 are comparable to the photograph in Figure 1.2. Both figures show how the parts of the clouds that directly cover the sun are brightly lit, while the surrounding clouds have bright contours with darker cores.

For the higher order of scattering, the result of using our approximation in the light function looks believable when the first order of scattering also contributes (as in Figure 3.9), but in Figure 3.8 the angle to the sun is around 180° and the phase function becomes negligible. The edges of the clouds in this image are darker, giving the clouds some distinct contours, but the cores—where the higher order light scattering is approximated—have a heavily saturated white look and most details are gone.

# Chapter 5
# Conclusion

Our implementation shows that there is much computational time to be saved when using ray marchers, mainly by presuming the circumstances that concern the scene that is to be rendered. By presuming the location of the media that is to be ray marched, the largest optimisation could be done.

The second ray marcher used for the lighting calculation is necessary for realist shadows, however it does have a large impact on the performance and other ways of calculating the light could have been tested if time had allowed.

The noise based cloud texture work well for the cloud forming, both as a satisfactory visual result and the non-complex implementation allows for parts of it to be either implemented on the shader or precalculated and saved to textures. This helps in tuning for optimal balance between memory bandwidth usage and computation time. For the cloud forming we only scratched the surface of the possible ways this could be done. We generated the 3D texture by "carving" out the clouds from noise textures, but the 3D textures could have been generated by using a fluid solver or maybe by constructing the clouds by combining smaller cloud pieces (tiles) and utilising Wang tiles.

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

# Bibliography


[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. pp. 11–26, 31, 37, 479.

[2] Edward Angel and Dave Shreiner. *Interactive Computer Graphics, A Top-Down Approach with Shader-Based OpenGL®*. Pearson Education, 6th edition, 2012. pp. 36–37.

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

[4] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. *Interactive multiple anisotropic scattering in clouds*. ACM Symposium on Interactive 3D Graphics and Games (I3D), 2008.

[5] Keenan Crane, Ignacio Llamas, and Sarah Tariq. *GPU Gems 3*. Pearson Education, 2008. pp. 633–675.

[6] Mark J. Harris and Anselmo Lastra. *Real-Time Cloud Rendering*. EUROGRAPHICS 2001, 2001.

[7] Brocken Inaglory. `https://upload.wikimedia.org/wikipedia/commons/6/60/Fogbow_glory_spectre_bridge_edit_1.jpg`. [Accessed: 2016-05-17].

[8] Henrik Wann Jensen. *Realistic Images Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001. pp. 115–121.

[9] Philip Laven. *MiePlot*. `http://www.philiplaven.com/mieplot.htm`. [Accessed: 2016-03-22].

[10] Basil John Mason. *Clouds, Rain and Rainmaking*. Cambridge University Press, 1962. pp. 1, 9, 12–15, 35.

[11] Ken Perlin. *Improved Noise reference implementation*. http://mrl.nyu.edu/~perlin/noise/, 2002. [Accessed: 2016-05-04].

[12] Ken Perlin. *Improving Noise*. http://mrl.nyu.edu/~perlin/paper445.pdf, 2002. [Accessed: 2016-05-04].

[13] Andrew Schneider and Nathan Vos. *The Real-Time Volumetric Cloudscapes of Horizon: Zero Dawn*. SIGGRAPH 2015 Advances in Real-Time Rendering, 2015. http://advances.realtimerendering.com/s2015/The%20Real-time%20Volumetric%20Cloudscapes%20of%20Horizon%20-%20Zero%20Dawn%20-%20ARTR.pdf. [Accessed: 2016-02-02].

[14] Leonard Teo. *CG Science for Artists Part 2: The Real-Time Rendering Pipeline*. http://www.cgchannel.com/2010/11/cg-science-for-artists-part-2-the-real-time-rendering-pipeline/, 2010. [Accessed: 2016-03-09].

[15] Wikipedia. *Beer-Lambert — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Beer%E2%80%93Lambert_law&oldid=718072472, 2016. [Accessed 2016-05-18].

[16] Steven Worley. *A cellular texture basis function*. SIGGRAPH '96 Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 1996. pp. 291–294.

[17] Íñigo Quílez. *Advanced Perlin Noise*. http://www.iquilezles.org/www/articles/morenoise/morenoise.htm. [Accessed: 2016-06-18].

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-06-03

**EXAMENSARBETE** Real-Time Rendering of Volumetric Clouds (Ray Marching Noise Based 3D Textures)
**STUDENT** Rikard Olajos
**HANDLEDARE** Michael Doggett (LTH)
**EXAMINATOR** Flavius Gruian (LTH)

# Realtidsrendering av volymetriska moln

POPULÄRVETENSKAPLIG SAMMANFATTNING **Rikard Olajos**

Att återskapa volymetriska moln digitalt är en kostsam process, räknat i processortid, och kan vara svårt att genomföra i realtid. Men behovet för realistiska moln i 3D applikationer finns och lösningen involverar en balansgång mellan vad som praktiskt går att göra och vad som ser verklighetstroget ut.

På grund av sin enkla implementering och snabba rendering har det länge varit vanligt att använda det som kallas *skyboxes* i spel för att representera en himmel i 3D-världen. En skybox består av sex sidor, med bilder på himmeln i olika riktningar, som omger betraktaren och ger en illusion av en himmel som är placerad oändligt långt bort. En sådan lösning fungerar när betraktaren inte förväntas röra sig över stora ytor eller röra sig upp i himmeln nära molnen, exempelvis som i bilspel. Men i *open world*-spel där spelaren kan röra sig mer eller mindre fritt i 3D-världen, ger en skybox ingen känsla av att man förflyttar sig. Man kan återskapa moln genom att bygga upp dem av bilder (2D-texturer) och som fysiskt existerar i 3D-världen. Dessa moln ger ett mer trovärdigt intryck men eftersom de är skapade av ”platta” bilder kan det uppstå konstiga bieffekter så som rotationer när man passerar rakt under eller över ett moln. För att undvika dessa bieffekter som 2D-baserade moln för med sig, måste molnen byggas upp med 3D-texturer.

Detta arbete presenterar ett enkelt sätt att skapa dessa 3D-texturer samt jämför och diskuterar olika sätt att rendera dem. 3D-texturerna bygger på brusfunktioner som kombineras med andra funktioner för att skapa de drag som karaktäriserar moln. Dessa funktioner ser till exempel till att ge molnen en relativt platt undersida medan ovansidan kan tillåtas resa sig i tornliknande strukturer som ett resultat av den turbulens och konvektion som sker i riktiga moln. Bildserien ovan visar ett tvärsnitt av 3D-texturer som visar arbetsgången och uppbyggnaden av molnen. Det finns inget naturligt sätt att återge 3D-texturerna direkt, utan de måste samplas genom att stega från varje pixel på användarens bildskärm ut i scenen för att avgöra hur mycket moln som ska synas för en viss pixel. Denna metod kallas *ray marching*. Genom att analysera 3D-texturerna innan de placeras i scenen, kan en lågupplöst struktur skapas och användas för minska antalet steg som måste tas. Detta sparar mycket processorkraft och gör det möjligt att rendera molnen i realtid.