UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science curriculum

Siim Raudsepp

Volumetric Fog Rendering

Bachelor's thesis (9 ECT)

Supervisor: Jaanus Jaggo, MSc

Tartu 2018

Volumetric Fog Rendering

Abstract:

The aim of this bachelor's thesis is to describe the physical behavior of fog in real life and the algorithm for implementing fog in computer graphics applications. An implementation of the volumetric fog algorithm written in the Unity game engine is also provided. The performance of the implementation is evaluated using benchmarks, including an analysis of the results. Additionally, some suggestions are made to improve the volumetric fog rendering in the future.

Keywords:

Computer graphics, fog, volumetrics, lighting

CERCS: P170: Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Volumeetrilise udu renderdamine

Lühikokkuvõte:

Käesoleva bakalaureusetöö eesmärgiks on kirjeldada udu füüsikalist käitumist looduses ja koostada algoritm udu implementeerimiseks arvutigraafika rakendustes. Töö raames on koostatud volumeetrilist udu renderdav rakendus Unity mängumootoris. Töös hinnatakse loodud rakenduse jõudlust ning analüüsitakse tulemusi. Samuti tuuakse töös ettepanekuid volumeetrilise udu renderdamise täiustamiseks tulevikus.

Võtmesõnad:

Arvutigraafika, udu, volumeetria, valgustus

CERCS: P170: Computer science, numerical analysis, systems, control

Table of Contents

1. Introduction

As the computational power of graphics cards increases, computer games can use more demanding and physically-based real-time rendering techniques to bring out the visual quality of those games. One such effect is fog rendering. In the past, fog has been mainly used to mask the shortcomings of computer hardware [1]. An example of this is the computer game Silent hill [2] [3]. Nowadays, fog can be realistically simulated and made to interact with light.

The objective of this thesis is to explain how fog behaves in real life and to provide an algorithm for rendering realistic fog in computer graphics. The thesis also comes with a description of the implementation of the algorithm.

The first chapter of this thesis explains the theory behind fog in real life. The second chapter describes the algorithm for rendering volumetric fog. The third chapter focuses on describing the implementation of the algorithm. The final chapter discusses the results and contains benchmarks of the author's implementation and gives some suggestions on how to improve the algorithm in the future.

1.1 Fog rendering techniques

Historically, the render distance in video games has been low, because computers were not as powerful as they are today. To save performance, the camera's maximum rendering distance was set closer to the camera. By doing this, most of the scene geometry farther away would be clipped by the camera's far plane and thus not rendered. This produced an effect known as "pop-in", where the objects suddenly appeared in the camera's view if the camera was close enough.

The solution to this was to fade each pixel on the screen to a predetermined color when it was farther than a fixed distance from the camera (a technique also known as depth fog). This made the popping effect go away and gave the scene a certain atmosphere. Figure 1 shows how depth was used in Silent Hill to make an impression that the game took place in a location covered by thick fog.

Figure 1. Depth fog in Silent Hill

An improvement to rendering depth fog is to add height based fog, shown in Figure 2. Height fog makes the fog look more physically correct than just using depth fog by simulating the gathering of fog particles near the ground. For that, the height fog uses the world space Y axis coordinate to reduce the fog thickness according to height.

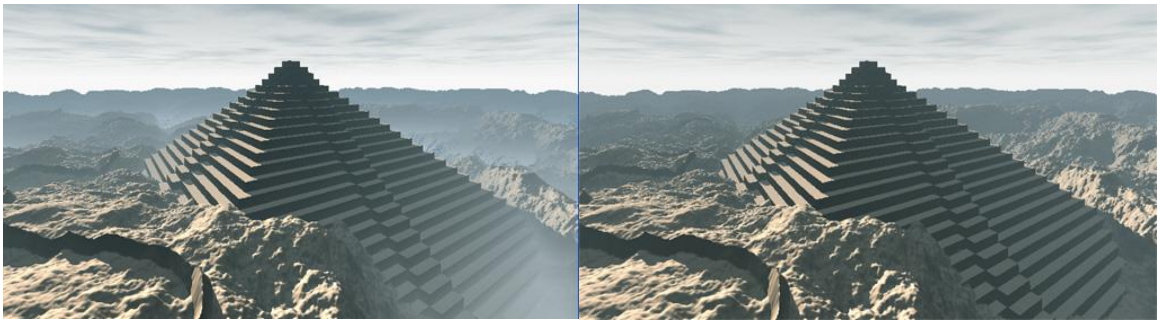

Figure 2. A scene with added height fog (left) and distance fog without height fog (right) [1]

The advantage of height and depth fog is that they are cheap to compute and give rather good-looking results. The disadvantage of these methods is that they have uniform fog density, which means that they cannot be animated. In addition, the height and depth fog are calculated only once per pixel thus they cannot be used to represent light passing through them.

A solution to the animation problem is to use billboards. A billboard is a quad with a 2D texture placed on it that rotates so that it is always facing towards the camera. By using a billboard with a semi-transparent fog texture placed in a scene, realistic looking animated fog can be achieved by scrolling the texture in some axis. The downside to this method is that whenever another surface intersects with a billboard, it produces a hard edge, which

[1] http://iquilezles.org/www/articles/fog/fog.htm

breaks the immersion. In Figure 3 the picture on the left has a sharp intersection between the ground geometry and the billboard. This can be solved by sampling the depth buffer and adjusting the opacity of the billboard according to how far it is from the object behind it – this technique is also called soft particles.

Additionally, drawing multiple semi-transparent textures over each other causes a lot of pixel overdraw, which means that the color and alpha values of a pixel get overwritten many times. This might also decrease the rendering performance severely.

Figure 3. Without considering the depth of the camera, the fog texture on the left appears cut off. [2]

Even by using soft particles, billboards still cannot represent how light propagates through a volume. For this reason, a new technique called volumetric fog was created. Volumetric fog is used in computer graphics to simulate how fog particles interact with light in real life.

[2] http://blog.wolfire.com/2010/04/Soft-Particles

2. Volumetric Fog Theory

It is not feasible to simulate every fog particle separately. Instead, volumetric fog estimates the density of fog particles in a relatively small space. The density is then used to calculate the physical interaction of the fog particles with the incoming light, namely transmission, absorption, and scattering of the light. This simulation is performed only for the world regions visible to the camera; therefore, the space under observation is the camera view volume. The camera view volume is split into uniform chunks, where the X and Y dimensions of a chunk are equal to the size of a pixel and the Z dimension of a chunk is calculated according to the size of the volume in the Z axis.

Figure 4 shows that light entering a volume can be either out-scattered, in-scattered, absorbed or transmitted [4].

- Out-scattering is the scattering of light back towards the light source.
- In-scattering is the scattering of light towards the viewer.
- Absorption is the loss of light energy upon interaction with a particle.
- Transmittance shows how much light energy is transmitted after all the other processes have taken place.

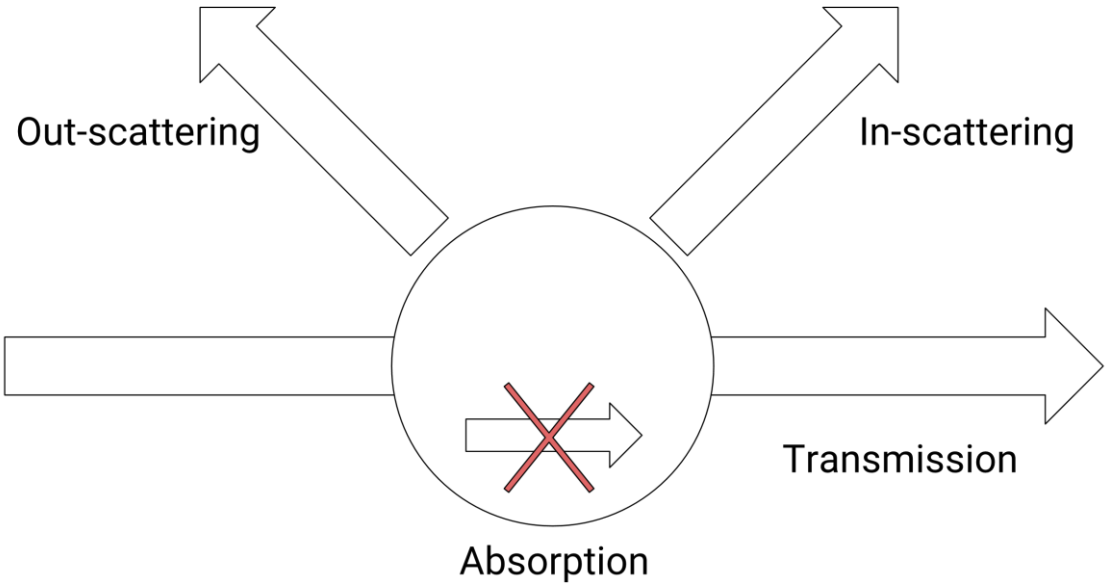

Figure 4. Different ways how light can interact with a particle.

According to Wroński [5], this process can be statistically modeled for each point of the volume using Equation 1.

$$L_{incoming} = L_{transmitted} + L_{absorbed} + L_{scattered}, \quad (1)$$

where $L_{transmitted}$ is the transmittance of light, $L_{absorbed}$ is the absorption and $L_{scattered}$ is the sum of in-scattered and out-scattered light.

2.1 Atmospheric Scattering

Since the Earth is surrounded by an atmosphere, the light particles coming from the sun and entering the eye of the viewer must interact with the atmosphere first. This means that some of the light gets scattered into the surrounding medium. However, the particles found in the atmosphere don't scatter all wavelengths of light equally: shorter wavelengths are scattered more. Because the colder colors also have shorter wavelengths, they get scattered more and this is the reason why the sky is blue [6]. Figure 5 shows the color of the atmosphere during a clear day.

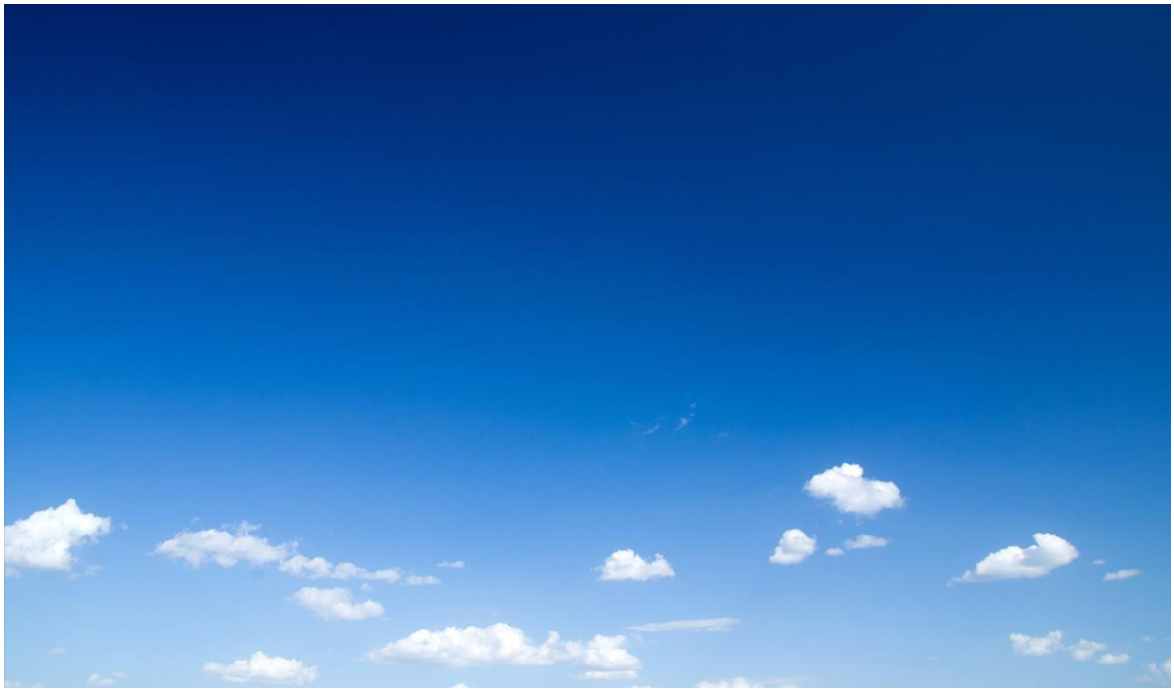

Figure 5. Blue sky due to scattering of light in the atmosphere[3]

Wroński [5] says that the amount of light participating in the scattering process can be different. For example, there exists Rayleigh scattering. It is the scattering of low wavelength particles in the atmosphere and is responsible for the blue color of the atmosphere [7]. In computer graphics, phase functions are used to describe the angular distribution of light scattered in each direction [8].

$$p(\theta, g) = \frac{3 * \left(1 + cos^2(\theta)\right)}{16 * \pi} \quad (2)$$

[3] http://airan.ee/est/wp-content/uploads/blue-sky-clouds.jpg

Equation 2 defines the function for calculating the Rayleigh phase function, where θ represents the angle between the direction of light and the direction of the viewer and $g \in [-1, 1]$ represents anisotropy. Anisotropy describes the directional dependency of the scattered light, which means that light scatters more in one direction [4].

There also exists Mie scattering, which is the scattering of bigger wavelength particles [4]. Mie scattering is too expensive to compute in real-time and thus a phase function is used to represent Mie scattering. The most common phase function used to approximate Mie scattering is the Henyey-Greenstein phase function. [8] [9]

Equation 3 defines the function for calculating the value of Henyey-Greenstein phase function. The parameters are same as used for the Rayleigh phase function.

$$p(\theta, g) = \frac{1 - g^2}{4\pi * (1 + g^2 - 2 * g * \cos(\theta))^{\frac{3}{2}}} \tag{3}$$

According to Cornette and Shanks [10], the Henyey-Greenstein phase function does not consider small particles illuminated by unpolarized light. In their paper, the authors propose of a more physically-based phase function, which has a similar form to the Henyey-Greenstein phase function. This means the Cornette-Shanks phase function can be easily used as a replacement for the Henyey-Greenstein phase function. Equation 4 defines the Cornette-Shanks phase function using the same parameters as the Henyey-Greenstein phase function and the Rayleigh phase function.

$$p(\theta, g) = \frac{3(1 - g^2) * \left(1 + \cos^2(\theta)\right)}{2 * (2 + g^2) * (1 + g^2 - 2 * g * \cos^2(\theta))^{\frac{3}{2}}} \tag{4}$$

To model the transmittance of incoming light, the Beer-Lambert law is used. [5] The Beer-Lambert law states that light transmittance is exponential to the distance traveled by light inside a medium.

$$T\ (A \rightarrow B) = e^{-\int_A^B \beta e(x)\, dx} \tag{5}$$

Equation 5 defines the function used to calculate Beer-Lambert law. βe is the sum of scattering and absorption coefficients. The law is used to calculate the transmittance of light energy at each point of the participating medium.

3. Algorithm for Rendering Volumetric Fog

The algorithm for rendering volumetric fog consists of 5 steps:

1) sampling the noise,
2) sampling the shadow map,
3) adding lighting,
4) applying blur to the fog,
5) blending and rendering to the screen.

3.1 Noise

In real life, fog does not have uniform density: some areas of the fog volume are denser and others are sparser. To mimic this characteristic in computer graphics, different noise generation algorithms are used, also known as noise functions. Noise functions are functions that return a continuous value in response to an input given to them. Noise functions are deterministic, but they still have some structure to them, making them perfect for representing fog in computer graphics.

A common way to create procedural textures is to precompute the noise using noise functions and save it to a texture, reducing the overall performance cost. Multiple noise textures can also be combined to get interesting effects like variable fog densities. Another advantage of noise functions is that they can be precomputed and stored in the computer's memory, reducing the overall performance cost of the algorithm. The noise texture used for the algorithm can be seen in Figure 6.

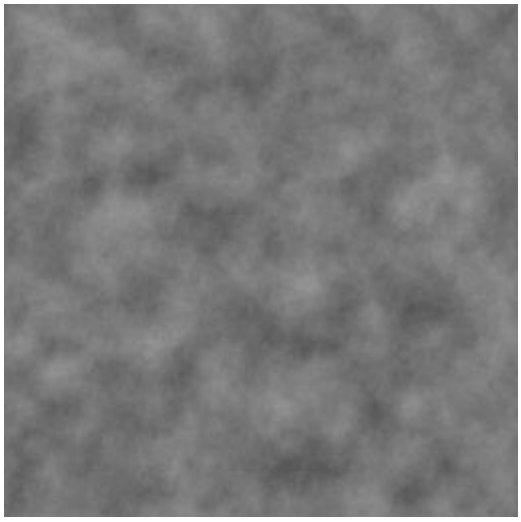

Figure 6. The noise texture the fog is sampled from[4]

[4] https://www.filterforge.com/filters/231.jpg

3.2 Sampling the noise

A 2D texture with precomputed noise values in the range of 0 to 1 is used to calculate the fog density at each point of the volume. The colors of a fog texture can be interpreted in different ways, but in this thesis, they represent fog density at some point in the fog. A value of 1 represents complete fogging at that point (transmittance is 0), meaning that nothing can be seen through it. A value of 0 represents no fogging at that point (transmittance is 1). Values between 0 and 1 get increasingly denser, meaning that less and less of the geometry behind the fog can be seen. The value of a sample is also based on height, meaning that fog farther from the reference level will be less dense. The fog density is then used to calculate how much light interacts with that particle. This is done by shooting rays through the volume and accumulating the result along the ray.

3.3 Sampling the shadow map

In this part of the algorithm, the locations of the shadows are calculated. For this, a shadow map is sampled so that the fog algorithm can decide if a point in the volume is in shadow. The result is then saved into a texture and used in the next parts of the algorithm. This step is responsible for the light shafts that represent the edges of shadows.

3.4 Adding lighting

In this part of the algorithm, the extinction, scattering and transmittance are calculated. The extinction values are constant for each point in the volume, so it can be represented as a coefficient multiplied by the fog density. The scattering value is calculated by summing up the values of the Cornette-Shanks phase function and Rayleigh phase function. The transmittance is calculated by applying Beer's law to the sample.

3.5 Applying blur to the fog

During light transport, some of the light gets scattered into the surrounding medium in real life, which creates a hazy effect to the fog. In computer graphics, this phenomenon is simulated by using blur. The blur effect is created by taking the values of surrounding pixels on the screen and interpolating the color values between them. As a result, a color of a pixel is now the weighted average of nearby pixels.

3.6 Blending and rendering to the screen

The final step of the algorithm is to blend the fog with the existing scene geometry. It is done by sampling the surface color and additively blending it with the fog color. The transmittance of the fog is contained in the fog textures alpha value. The lower the transmittance, the denser the fog is at that point, which means that less of the background geometry is visible to the camera.

4. Implementation

The algorithm is implemented in the Unity game engine[5]. In Unity, a C# script is used for setting up the necessary context, passing values to the shaders and storing the intermediate results in textures. Shaders are used to render the fog, add blur and blend it with existing scene geometry.

4.1 Architecture

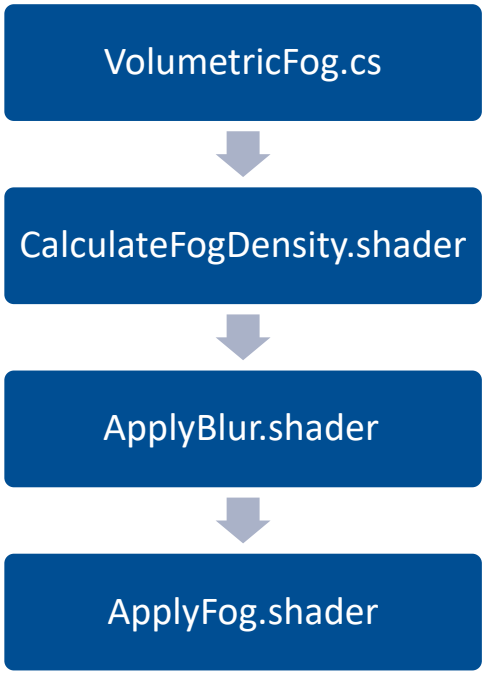

Figure 7. The architecture of the program used for rendering volumetric fog.

The architecture of the fog implementation in Unity can be seen in Figure 7. First, the C# script VolumetricFog.cs takes the parameters and sends them to the shaders. All the shaders are post-effect shaders, meaning that they change the pixel colors of an already rendered scene. The fog is rendered using the CalculateFogDensity.shader script. The result is stored in a texture and used in the ApplyBlur.shader script. This script is run many times to get an increasingly blurrier fog effect. Finally, the previous result is used in the ApplyFog.shader script to blend the fog with the scene geometry. After each frame, the render textures used are temporarily freed so they do not take up memory once they are not needed anymore.

4.2 Raymarching

Integrating the fog volume is done by casting a ray for each pixel of the screen and during multiple iterations, moving along the ray by a pre-calculated step size. The ray is then extended until transmittance reaches 0 or a maximum predefined number of steps is reached. This technique is also known as raymarching. Raymarching is used to accumulate fog density for each pixel separately.

```
float rayMarch(steps, direction, startPos){
```

[5] https://unity3d.com/

```
   // calculate the pixel depth using the uv of the pixel and the depth texture
   depth = calculate_pixel_depth();

   // maps the depth value to range [0,1]
   depth = linearise_depth(depth);

   // get the world space coordinates of the pixel
   endPos = viewPos_to_worldPos(depth);

   rayLength = length(endPos - startPos);

   // divide the steps evenly along the ray length
   stepSize = rayLength/steps;

   result = 0;

   // Start marching from the camera position
   currentPos = startPos;

   for(i = 0; i < steps; i++){

      currentResult = 0;

      noise = sample_noise(currentPos);

      currentResult = calculate_shadows_and_lighting(currentPos, noise);

      // Add current result to the overall result
      result += currentResult;

      // Extend the ray by a step in the ray direction
      currentPos += direction * stepSize;
   }

   return result;
}
```

Figure 8. The pseudocode used for raymarching

Figure 8 shows the pseudocode that is used by the CalculateFogDensity shader to accumulate fog density and calculate lighting for each pixel.

4.3 Sampling the shadow map and adding lighting

To sample from the shadow map, it first needs to be created and stored in a texture. Unity generates the shadow maps automatically for the scene, but storing it to a texture needs to be done manually. This is done by creating a command buffer and setting it to execute after the shadow map has been created. A command buffer is a buffer used to hold a list of rendering commands, which can be set to execute at various points during the scene rendering, light rendering or be executed immediately [11].

```
void AddLightCommandBuffer()
{
   // create a command buffer and give it a global identifier
   _AfterShadowPass = new CommandBuffer {name = "Volumetric Fog ShadowMap"};

   // store the result in the shader texture "ShadowMap"
   _AfterShadowPass.SetGlobalTexture("ShadowMap",
          new RenderTargetIdentifier(BuiltinRenderTextureType.CurrentActive));

   Light sunLight = SunLight.GetComponent<Light>();

   if (sunLight)
   {
     // Add the command buffer to the light and set it to execute after the
     // shadow map has been generated by Unity
      sunLight.AddCommandBuffer(LightEvent.AfterShadowMap, _AfterShadowPass);
   }
}
```

Figure 9. The code to set up a command buffer[6]

Figure 9 shows the code required to create a command buffer and attach it to a light. First, a new command buffer is created. After that, a render texture is created with the same parameters as the one currently being rendered to. This command buffer is then set to execute after the shadow map has been generated, saving the shadow map to a texture named "ShadowMap".

Unity uses Cascading Shadow Maps (CSMs) [12], which means that different parts of the camera view frustum have different shadow map resolutions. This technique is useful because usually the viewer can see details closer to them more clearly, but details farther away are harder to distinguish. The technique also increases the performance of using shadow maps because cascades farther away can be calculated with a smaller resolution.

[6] https://interplayoflight.wordpress.com/2015/07/03/adventures-in-postprocessing-with-unity/

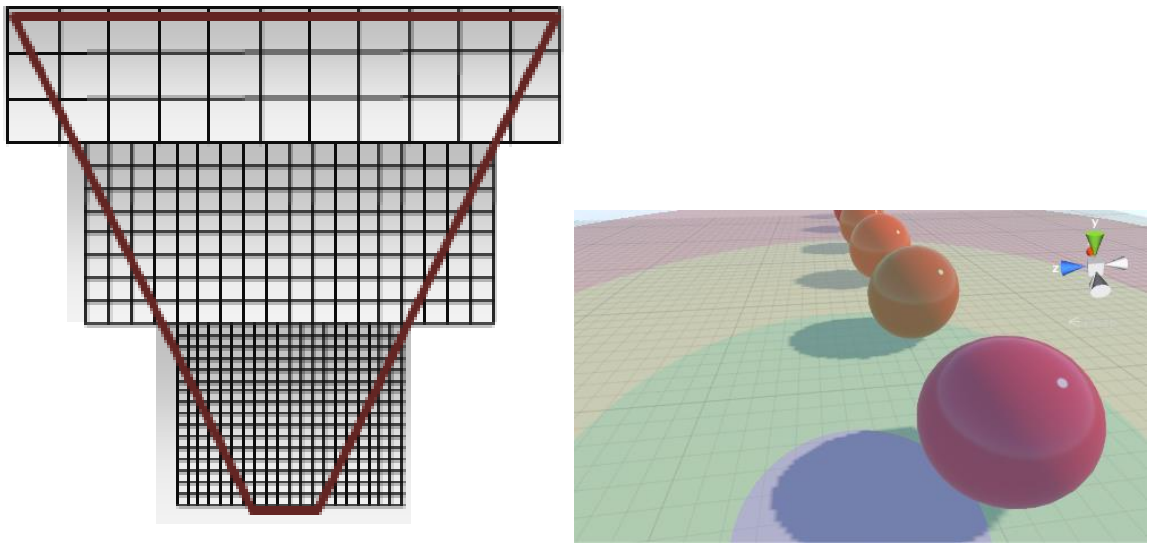

Figure 10. A top-down representation of a cascaded shadow map inside the camera view frustum[7] (left) and a cascaded shadow map projected to the ground surface in Unity[8] (right).

Figure 10 shows that cascades closer to the camera have more resolution, while the cascades far away have a noticeably smaller resolution. On the right, the cascades are shown as seen in Unity, where each color represents a different cascade. The distance of each cascade from the camera can be adjusted.

```
fixed4 getCascadeWeights(float depth){

   float4 zNear = float4(depth >= _LightSplitsNear);
   float4 zFar = float4(depth < _LightSplitsFar);
   float4 weights = zNear * zFar;

   return weights;
}
```

Figure 11. The code used to sample cascade weights

According to the depth value of the current pixel, the world space coordinates of the pixel are calculated. The code in Figure 11 is used to select all the cascades that are between the cameras near and far plane.

```
fixed4 getShadowCoord(float4 worldPos, float4 weights){

   float3 shadowCoord = float3(0,0,0);

   if(weights[0] == 1){
      shadowCoord += mul(unity_WorldToShadow[0], worldPos).xyz;
   }
   if(weights[1] == 1){
      shadowCoord += mul(unity_WorldToShadow[1], worldPos).xyz;
```

[7] https://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx
[8] https://docs.unity3d.com/Manual/DirLightShadows.html

```
    }
    if(weights[2] == 1){
        shadowCoord += mul(unity_WorldToShadow[2], worldPos).xyz;
    }
    if(weights[3] == 1){
        shadowCoord += mul(unity_WorldToShadow[3], worldPos).xyz;
    }

    return float4(shadowCoord,1);
}
```

Figure 12. The code used to get the correct shadow map coordinates

After that, the world space coordinates and the cascade weights can be used to calculate the coordinates for sampling the shadow map. Unity allows up to four cascades per shadow map and provides a matrix, unity_WorldToShadow, where rows are the transformations of each cascade in light space [13]. By multiplying this matrix with the world space coordinate of the current pixel, the correct light space coordinates for each cascade are found. The code for calculating the shadow map coordinate is shown in Figure 12.

```
float shadowTerm = UNITY_SAMPLE_SHADOW(ShadowMap, shadowCoord);
```

Figure 13. The code used to sample the shadow map

Using the found coordinate, the shadow map can be sampled to get the shadow term. The shadow term indicates whether or not the point on the ray is in shadow (Figure 13).

```
float3 fColor = lerp(_ShadowColor, litFogColor, shadowTerm + _AmbientFog);
```

Figure 14. The code used to mix two colors together based on the shadow term

Using the shadow term, the color of the particle at the current ray position is be calculated. At that point, ambient fog is also added to reduce the contrast between shadowed and non-shadowed areas (Figure 14).

The lighting for each step in the raymarching loop is calculated in 3 parts: extinction, transmittance and scattering (Figure 15).

- The extinction of light depends on the density of the fog at that point. It is multiplied by the extinction coefficient to allow the user to increase or decrease the extinction effect.
- The transmittance is found by applying the Beer-Lambert law to the extinction value. It is then multiplied by the extinction value, so areas further from the viewer will also transmit less light to the viewer.
- Scattering at each point of the volume is found by summing the values of the Mie scattering and the Rayleigh scattering.

```
float extinction = _ExtinctionCoef * fogDensity;

transmittance *= getBeerLaw(extinction, stepSize);

scattering = getRayleighScattering() + getMieScattering();
```

Figure 15. The code used to calculate extinction, transmittance and scattering

Finally, the color, scattering, step size and transmittance are all multiplied together to get the result of the current iteration. Once the loop terminates, the final color is outputted, where the alpha value of a pixel is equal to the transmittance variable value.

Figure 16. The fog after adding lighting

Figure 16 shows the result of this step, where the fog and the light rays shining through the trees are already present.

4.4 Gaussian blur and bilateral filtering

The blur effect was achieved by implementing a Gaussian blur. Gaussian blur is an image processing technique used to blur images. The blur effect is achieved by iterating through all the pixels on an image and for each pixel, averaging the color values of surrounding pixels based on the Gaussian distribution.

$$G(x,y) = \frac{1}{2\pi\sigma^2} * e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (6)$$

The contribution of each pixel is calculated using Equation 6, where x and y define the offset from the origin and σ is the standard deviation of the Gaussian distribution. [14]

Figure 17. The fog after adding Gaussian blur.

Figure 17 shows that after applying Gaussian blur, the resulting image is uniformly fuzzy. This can be fixed by using bilateral filtering. Bilateral filtering is a technique used to preserve the sharpness of edges in an image. It compares the intensity values between neighboring pixels. A greater intensity difference means that the pixel sampled is on the edge of the geometry and that pixel contributes less to the blur. [15]

Figure 18. The fog after adding bilateral filtering to the blur.

Figure 18 shows that by adding bilateral filtering, the edges of the geometry are sharp while the rest of the view remains blurred.

4.5 Blending the fog with existing scene geometry

```
// Sample fog texture
float4 fogSample = tex2Dlod(FogRendertargetLinear, float4(input.uv,0,0));

// Sample scene texture
float4 colorSample = tex2D(_MainTex, input.uv);

// blend samples together
float4 result = float4(colorSample.rgb * fogSample.a + fogSample,colorSample.a);

return result;
```

Figure 19. The code used to blend the scene geometry with the fog

Finally, the fog must be blended with the scene geometry. This is done by using additive alpha blending, shown on Figure 19. First, the fog texture and the color texture are sampled. After that, the samples are blended together and the result is returned.

Figure 20. The result after blending the fog with scene geometry

Figure 20 shows that colors from existing scene geometry and the skybox have been correctly blended with the volumetric fog. Colors from the existing scene geometry are contributed to the fog color.

5. Results

This chapter discusses the results of the volumetric fog implementation. Although this implementation can be optimized further and is more for the proof of concept, benchmark comparisons with another volumetric fog implementation, Aura[9], have also been made. Both the code and the assets for the application were hosted on GitHub[10] and are publicly available. Details on how to run the application and a link to the repository are given in Appendix 2.

The program, VolumetricFog.cs, is used to set up the necessary context for the shaders and to get the shadow map from Unity. All the parameters of the volumetric fog are also sent from this script to the specific shader that requires them. The parameters can be seen in Figure 21. More detailed descriptions of each parameter are given in Appendix 1.

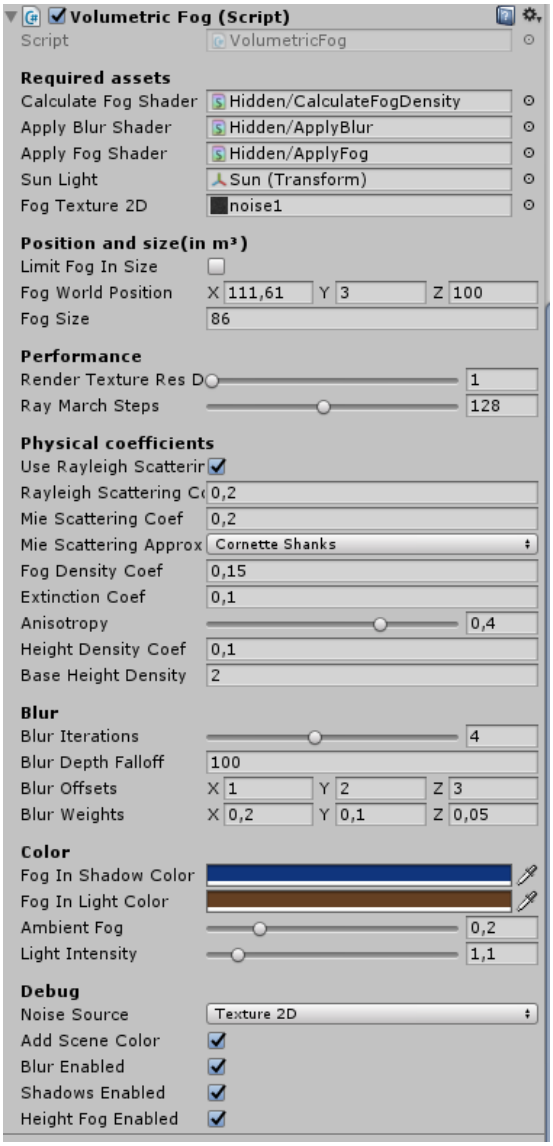

Figure 21. Tweakable values of the volumetric fog script.

[9] https://www.assetstore.unity3d.com/#!/content/111664

[10] https://github.com/

The shader programs were written in ShaderLab language, which is a declarative language that is used in Unity to describe the shader. The ShaderLab language wraps around the fragment and vertex shaders, which are written in HLSL/Cg shading language [16].

The main scene, “Forest”, was made using Unity’s built-in terrain system [17]. The terrain system allows to quickly create various landscapes and add custom textures to them. The foliage for the scene was provided by Nature Starter Kit 2 [18], a free asset package found from the Unity Asset Store[11].

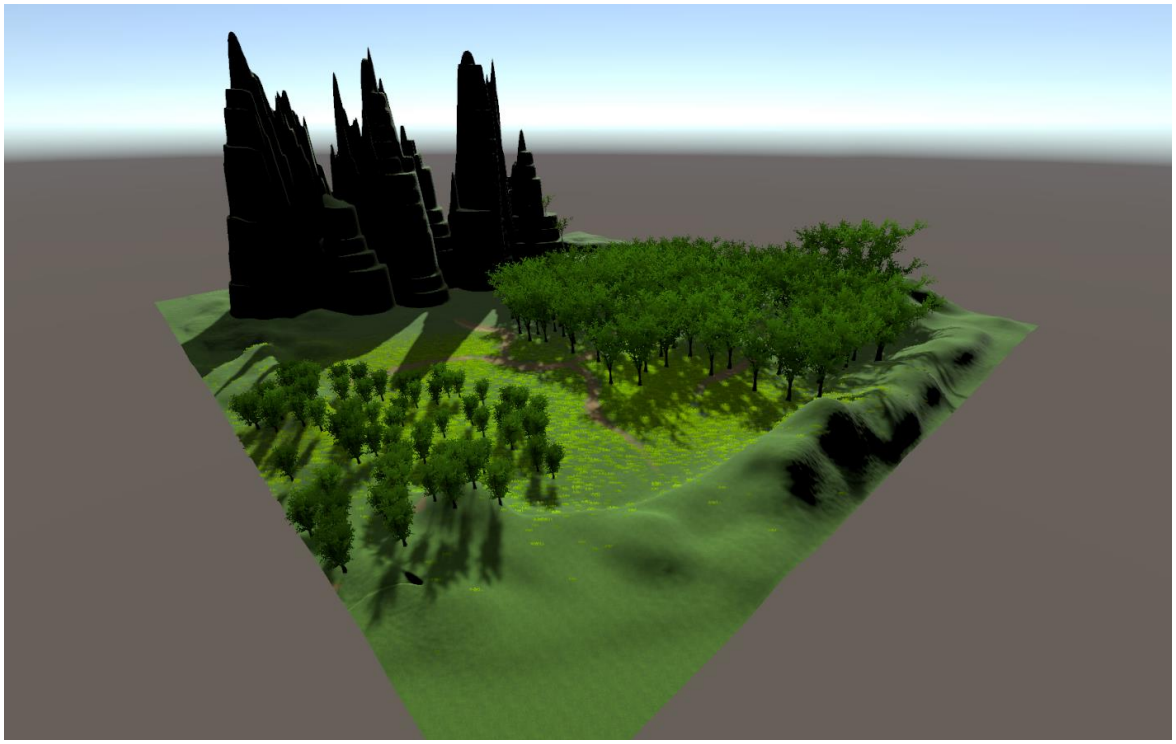

Figure 22. The “Forest” scene built for testing the volumetric fog effect.

Figure 22 shows the layout of the scene. The scene consists of 4 main areas: dense forest, sparse forest, open grassland and mountains.

5.1 Benchmarks

To evaluate the performance of this implementation, benchmarks were performed. A benchmark consists of a camera flying through the scene for 50 seconds. Before the benchmark, 2 seconds of warmup time is given to the computer to let it start rendering and stabilize the frame rendering time. For each frame, the frame time is measured and later averaged for the whole run. The standard deviation of each run is also found. A resolution of 1920x1080p is used for all the benchmarks and the values used for the volumetric fog script are the same as in Figure 21, unless stated otherwise.

The benchmarks were performed in Unity version 2017.4.1f1. The computer used for running the benchmark has the following specifications:

[11] https://assetstore.unity.com/

- CPU: Intel Core i5 4670K, running at 4.1GHz
- GPU: Nvidia GeForce GTX 1060 6GB
- RAM: 12GB DDR3, running at 1600Mhz
- OS: Windows 10 Pro 64-bit

5.1.1 Performance without volumetric fog

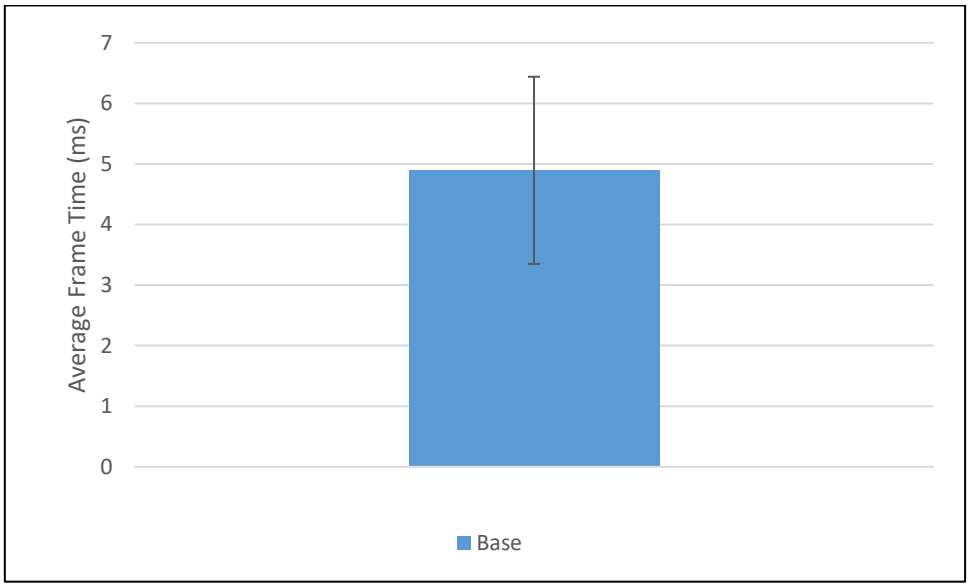

Figure 23. Average frame time of the benchmark with the volumetric fog effect turned off.

First, a benchmark was done with the volumetric fog effect turned off. Its purpose is to get an idea of the rendering cost of the effect. Figure 23 shows that the average frame time of the benchmark without the effect is almost 5 milliseconds.

5.1.2 Performance comparison on the resolution used

To test how the algorithm scales with different resolutions, the benchmark with the fog effect enabled was run for 3 times with resolution set to 640x480(480p), 1280x720(720p) and 1920x1080(1080p) pixels, respectively.

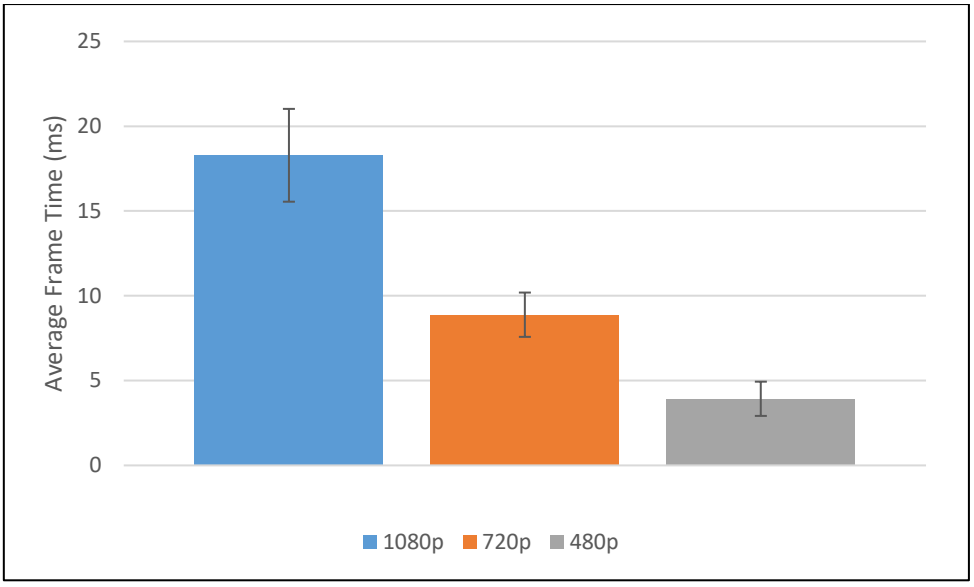

Figure 24. Average frame time of the screen resolution benchmark

Figure 24 shows that increasing the resolution affects the performance of the effect almost linearly. 1080p has about 44% more pixels to render than 720p and the effect takes about 44% longer to render. The same pixel to millisecond ratio can be seen when comparing 720p to 480p.

5.1.3 Performance comparison on the noise source used

To evaluate the performance of each noise source present in the application, the benchmark was performed 3 times, each time with a different noise source.

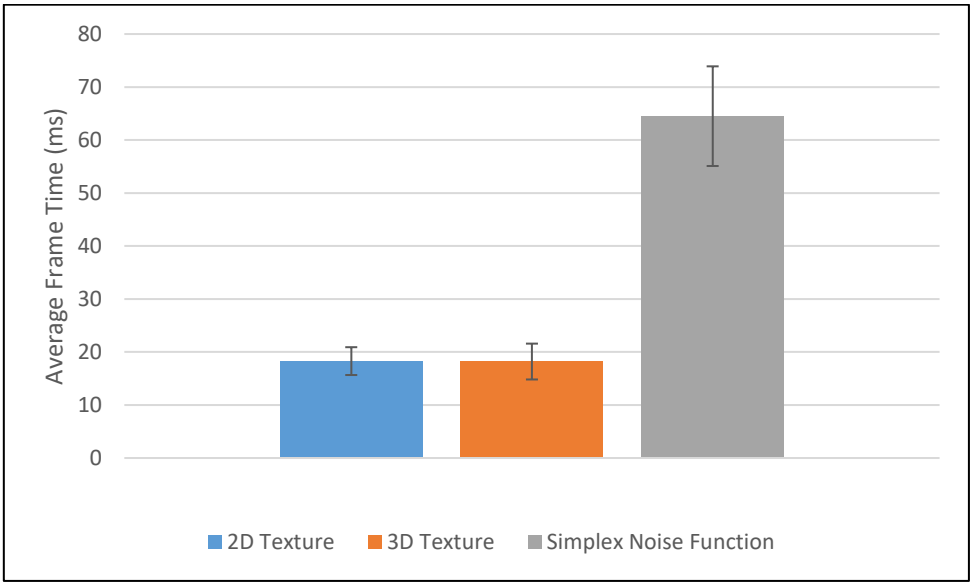

Figure 25. Average frame time of the noise source benchmark.

Figure 25 shows that generating the noise each frame using the Simplex noise function affects performance severely (an average frame time over 64 milliseconds). However, there is almost no difference between sampling the noise from a 3D or 2D texture (both taking about 18 milliseconds per frame to render).

5.1.4 Performance comparison on the number of raymarching samples

To compare the performance impact of the number of raymarching samples used, the benchmark was run for 3 times using the settings in Figure 21. The volumetric fog effect was rendered using 64, 128 and 256 raymarching samples, respectively.

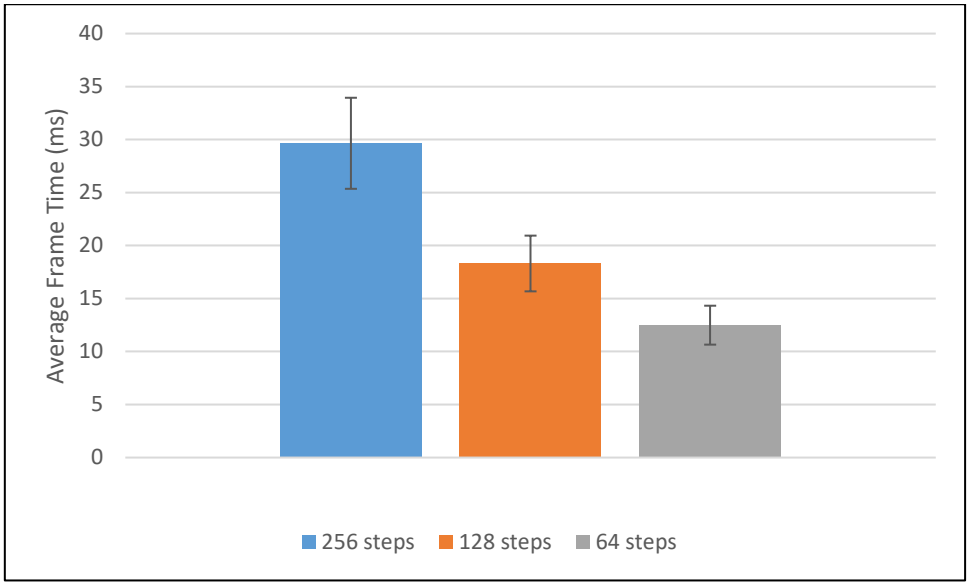

Figure 26. Average frame time of the raymarching samples benchmark.

Figure 26 shows that increasing the number of raymarching samples affects the performance. Using 256 raymarching steps, the average frame time is about 30 milliseconds. By reducing the raymarching steps to 128, the cost of rendering is reduced to 18 milliseconds per frame. Halving the raymarching steps again, to 64 steps, reduces the cost to 12 milliseconds per frame.

5.1.5 Performance comparison versus Aura volumetric fog

Finally, the performance of this implementation using the base settings (Figure 21) was tested against the performance of another volumetric fog implementation, Aura. Although Aura advertises itself as a volumetric lighting solution, the main effect implemented is volumetric fog. The settings of Aura were kept as close as possible to this implementation and the same benchmark in the same scene was used to compare Aura to this implementation.

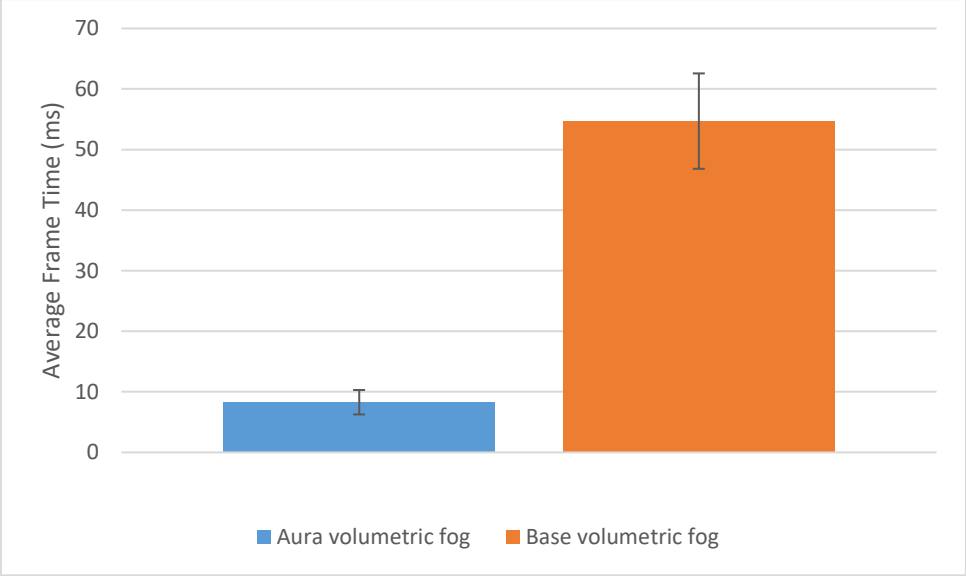

Figure 27. Comparison of the base volumetric fog implementation and Aura.

Figure 27 shows that the performance of Aura exceeds the reference implementation's performance greatly (almost 7x faster than the reference implementation) while delivering better visual results.

5.2 Visual results

This section showcases the visual results of the created application. The colors used were chosen artistically to mimic the atmosphere during the sunrise. The shadow color has a colder tone and the fog color has a warmer tone.

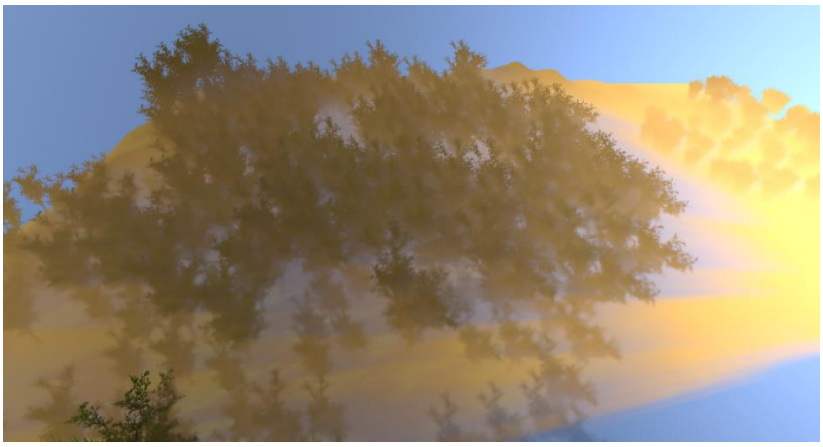

Figure 28. Volumetric fog as seen from above

Figure 28 shows the scene with the camera positioned higher from the fog and pointed towards the ground. The figure also shows that trees far away from the camera are drowned in the fog.

Figure 29. Volumetric fog as seen from the ground

Figure 29 shows the scene with the camera positioned on the ground, looking towards the sun. Crepuscular rays can also be seen, which are created by the sunlight that is shining through the trees.

Figure 30. Fog in shadowed areas.

Figure 30 shows the scene with the camera positioned on the ground, looking at the shadows cast by the trees. Almost all the fog that can be seen is ambient, since no sunlight is shining through the fog directly.

5.3 Future improvements

This chapter discusses the changes that could be done in the future to improve the performance and the visual quality of the current implementation of the volumetric fog.

5.3.1 Interleaved sampling

In the current implementation, marching a ray for each pixel on the screen uses a high number of steps. This increases the visual quality of the result, but severely decreases the performance. To reduce the number of rays that are marched each frame, interleaved sampling could be used. Interleaved sampling is the sampling of textures using an offset from the center [19]. It works by dividing the screen into evenly sized grids, each grid consisting of

a set number of neighboring pixels. Since the color values of volumetric lighting and surfaces affected by lighting are similar between neighboring pixels [20], the color result of a pixel can be used for neighboring pixels in the same grid as well.

5.3.2 Blending the effect with the skybox

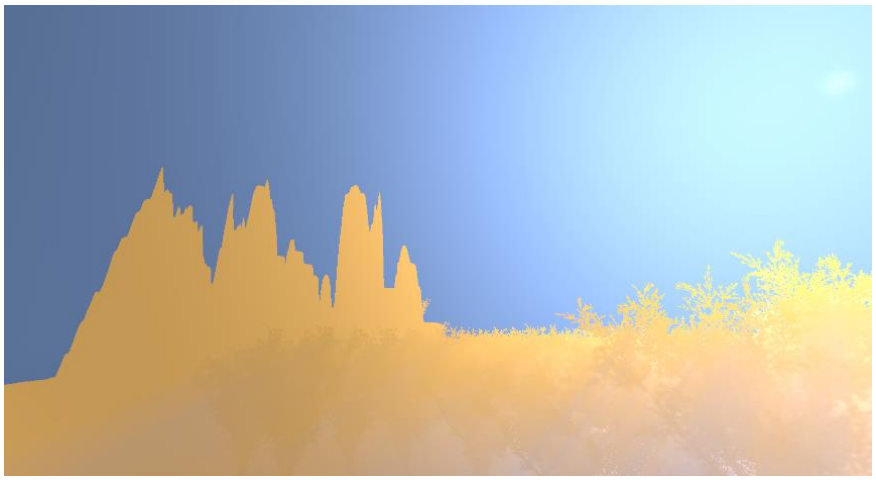

Figure 31. The fog effect can only be seen on opaque geometry.

Currently, the fog effect is only rendered on top of opaque geometry. This shortcoming is even more visible when increasing the density of the fog such that the existing scene geometry blends into the fog (Figure 31). The volumetric fog effect does not render on top of the skybox because in Unity, the skybox is rendered after all opaque geometry[12]. One solution to this is to set the skybox color manually to match the fog color. Another solution to this would be to use a 3rd party asset, KinoFog[13]. KinoFog fades the color of each pixel to the color of the skybox depending on depth. By setting the skybox color according to the color of the fog near a point in the skybox, it is possible to blend the fog with the skybox.

[12] https://docs.unity3d.com/Manual/class-Skybox.html

[13] https://github.com/keijiro/KinoFog

6. Summary

Since today's graphics cards have gained a lot of computational power, they can now be used to render real-time realistic effects in computer graphics applications. One such effect is fog, which has been historically used to mask the low render distance. Nowadays, fog is mainly used to create a sense of an atmosphere. To meet this requirement, the fog must interact with light and the surrounding environment realistically. This is achieved by using a technique known as volumetric fog rendering.

In this thesis, a brief history of fog in computer graphics was described. It was also observed how the fog behaves in real life and how its physical behavior could be simulated in computer graphics. After that, an algorithm for rendering real-time volumetric fog was provided and a proof of concept application was created in the Unity game engine. To measure the performance of the implementation, benchmarks of the demo application were made. Although the current implementation was not as fast as other implementations, namely Aura[14], all the goals of this thesis were met successfully.

Finally, some improvements were suggested to lessen the impact of the effect on performance and to further improve the visual quality of the implementation. These serve as a good starting point for others to further develop the application and to improve the algorithm in the future.

[14] https://www.assetstore.unity3d.com/#!/content/111664

7. References

[1] M. M. Deza and E. Deza, in *Encyclopedia of Distances*, 2009, p. 513.

[2] "Silent Hill (video game)," [Online]. Available: https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Silent_Hill_(video_game).html. [Accessed May 2018].

[3] A. Hodgetts, "Team Silent and Combating Technical Limitations," 20 December 2017. [Online]. Available: https://airentertainment.biz/2017/12/20/team-silent-and-combating-technical-limitations. [Accessed May 2018].

[4] J.-C. Prunier, "Volume Rendering for Artists," 2017. [Online]. Available: https://www.scratchapixel.com/lessons/advanced-rendering/volume-rendering-for-artists.

[5] B. Wroński, "Volumetric Fog and Lightning," in *GPU Pro 6: Advanced Rendering Techniques*, 2015, pp. 217-242.

[6] "Scattering of light," [Online]. Available: http://www.atmo.arizona.edu/students/courselinks/spring08/atmo336s1/courses/fall13/atmo170a1s3/1S1P_stuff/scattering_of_light/scattering_of_light.html. [Accessed 18 April 2018].

[7] J.-C. Prunier, "Simulating the Colors of the Sky," [Online]. Available: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/simulating-sky. [Accessed May 2018].

[8] N. Vos, "Volumetric Light Effects in Killzone: Shadow Fall," in *GPU Pro 5*, 2014, pp. 127-146.

[9] A. Schneider, "Real-Time Volumetric Cloudscapes," in *GPU Pro 7: Advanced Rendering Techniques*, 2016, pp. 97-127.

[10] J. G. Shanks and W. M. Cornette, "Physically reasonable analytic expression for the single-scattering phase function," *Applied Optics*, vol. 31, no. 16, pp. 3152-3160, 1992.

[11] Unity Technologies, "Unity 2018.1 Documentation : Command buffer," Unity Technologies, 2018. [Online]. Available: https://docs.unity3d.com/ScriptReference/Rendering.CommandBuffer.html. [Accessed 13 May 2018].

[12] Unity Technologies, "Unity 2017.3 Documentation : Directional light shadows," [Online]. Available: https://docs.unity3d.com/Manual/DirLightShadows.html. [Accessed 11 April 2018].

[13] K. Anagnostou, "Interplay of light : Adventures in postprocessing with Unity," 3 July 2015. [Online]. Available: https://interplayoflight.wordpress.com/2015/07/03/adventures-in-postprocessing-with-unity. [Accessed March 2018].

[14] sonic0002, "Gaussian Blur Algorithm," 24 November 2012. [Online]. Available: https://www.pixelstech.net/article/1353768112-Gaussian-Blur-Algorithm. [Accessed 09 May 2018].

[15] S. Paris and F. Durand, "A Gentle Introduction to Bilateral Filtering and its Applications," 2007. [Online]. Available: https://www.cis.rit.edu/~cnspci/references/dip/filtering/paris2007.pdf. [Accessed 08 May 2018].

[16] Unity Technologies, "Unity 2018.1 Documentation : ShaderLab Syntax," [Online]. Available: https://docs.unity3d.com/Manual/SL-Shader.html . [Accessed May 2018].

[17] Unity Technologies, "Unity 2018.1 Documentation : Terrain Engine," [Online]. Available: https://docs.unity3d.com/Manual/script-Terrain.html . [Accessed May 2018].

[18] Shapes, "Unity Asset Store," [Online]. Available: https://assetstore.unity.com/packages/3d/environments/nature-starter-kit-2-52977. [Accessed May 2018].

[19] A. Keller and W. Heidrich, "Interleaved sampling," 2001. [Online]. Available: http://www.cs.ubc.ca/labs/imager/tr/2001/keller2001a/keller.2001a.pdf. [Accessed 06 May 2018].

[20] B. Tóth and T. Umenhoffer, "Real-time Volumetric Lighting in Participating Media," 2009. [Online]. Available: http://sirkan.iit.bme.hu/~szirmay/lightshaft.pdf. [Accessed 07 May 2018].

Appendix

I. Parameter descriptions of the VolumetricFog.cs script

Parameter name	*Description*
Calculate Fog Shader	The shader that calculates the fog density and color at each point of the volume.
Apply Blur Shader	The shader that applies blur to the texture produced by the calculate fog shader.
Apply Fog Shader	The shader that blends the fog together with the background geometry.
Sunlight	The game object of the directional light caster in the scene.
Fog Texture 2D	The texture that the noise is sampled from. This is also used for creating the 3D fog texture.
Limit Fog In Size	Toggles the size limiting of volumetric fog in the scene.
Fog World Position	The center coordinates of volumetric fog in the scene.
Fog Size	The size of volumetric fog in each axis calculated from the Fog World Position.
Render Texture Res Division	Determines the size of the render texture in comparison to the screen size.
Ray March Steps	The maximum number of steps that the algorithm uses to render the volumetric fog effect.
Use Rayleigh Scattering	Toggles the usage of Rayleigh scattering in the shaders on and off.
Rayleigh Scattering Coef	The coefficient of the Rayleigh scattering term.
Mie Scattering Coef	The coefficient of the Mie scattering term.
Mie Scattering Approximation	Determines the function that is used to approximate Mie scattering. Possible values are Henyey-Greenstein, Cornette-Shanks and Off.
Fog Density Coef	The coefficient that gets added to the fog value at each point in the volume.

Extinction Coef	The coefficient that determines how much light gets extinct at each point in the volume.
Anisotropy	Determines in which direction the light is mainly scattered. A value of -1 means that all the light is scattered back towards the light source and a value of 1 means that all the light is scattered towards the viewer.
Height Density Coef	Determines the rate of fog density falloff if height fog is enabled.
Base Height Density	Determines the density of volumetric fog in the bottom of the volume.
Blur Iterations	Determines how many times the blurring takes place.
Blur Depth Falloff	Determines the range around the viewer where no blur is added.
Blur Offsets	Determines the pixel offsets of each neighboring pixel when blurring the image.
Blur Weights	Determines the contribution of color from each neighboring pixel.
Fog In Shadow Color	The color of volumetric fog directly in shadow.
Fog In Light Color	The color of volumetric fog directly in light.
Ambient Fog	The amount of fog added to the shadowed areas of volumetric fog.
Light Intensity	Determines the intensity of the directional light caster.
Noise Source	Determines where the noise used in fog is sampled from. Possible values are Texture2D, Texture3D and Simplex Noise Function.
Add Scene Color	Toggles the blending of fog with the background on and off.
Blur Enabled	Toggles blur on and off.
Shadows Enabled	Toggles shadows on and off.
Height Fog Enabled	Toggles height based fog on and off.

II. Description of the application

Attached to this thesis is the demo application built to showcase the volumetric fog effect. To run the application, the executable (VolumetricFog.exe) should be executed. When running the program, it will first prompt to select the settings. After the desired settings are chosen, the play button must be pressed to start the application. The default controls for moving the camera are defined in Table 1.

Control name	*Description*
Mouse	Look around
W	Move forward
S	Move backward
A	Strafe left
D	Strafe right
Shift	Hold down while moving to move faster

Table 1. The controls of the application

The application is also hosted on GitHub[15]. Descriptions on how to download and run the source code in Unity are explained in the readme of the repository.

[15] https://github.com/SiiMeR/unity-volumetric-fog

III. License

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Siim Raudsepp**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
   Volumetric Fog Rendering,
   mille juhendaja on Jaanus Jaggo,
   1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.
3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, **14.05.2018**