DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Implementation of Volumetric Light Scattering in Unity

Felix Kosian

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Implementation of Volumetric Light Scattering in Unity

Implementation von Volumetrischer Lichtstreuung in Unity

Author:	Felix Kosian
Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	Prof. Dr. Rüdiger Westermann
Submission Date:	15.04.2020

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.04.2020 Felix Kosian

Acknowledgments

I wish to express my sincere thanks to the following people:

- My supervisor Prof. Dr. Rüdiger Westermann for letting me choose and supporting my topic
- Sebastian Lague for inspiring me with his coding adventure YouTube series
- My friends for supporting me with ideas, help with Blender and test reading this thesis

Abstract

This thesis briefly analyzes the physics of light scattering in air, describes and compares all commonly used mesh-based, post-processing and volumetric approaches to create the light scattering effect in real-time applications (games). Furthermore it explains how to implement the view frustum voxelization (froxel) approach with corresponding 3D textures and compute shaders in Unity to create god rays. This is the newest approach, which is used in many state of the art game engines. Different depth distributions are analyzed and future possibilities for optimizations are summarized.

Contents

1 Introduction

Volumetric light effects like fog and clouds add realism and atmospheric feeling to a scene. Especially the god ray effect is a fascinating phenomenon. The digital visualization of volumetric effects offers many approaches and is often computationally intensive. In the first part of this thesis we will analyze what volumetric light scattering is and how it is present in the real world. In the second part, we will gather and compare all commonly used methods of implementing this effect in games. In the third part we will implement a simple version of the newest approach (volumetric textures) in Unity with focus on visualizing every step of the algorithm. Finally we will compare the results of different depth distributions and take a look at possible optimizations.

Figure 1.1: Example of god rays in nature [Unk17]

2 Physics of Volumetric Light Scattering

Volumetric light scattering describes the effect of light traveling through participating media e.g. fog, dust and smoke. In reality the path of photons is influenced. The result is the visibility of light in 3D space.

Four effects take place for this phenomenon: absorption, emission, out-scattering and in-scattering [Jar08]. See figure 2.1. Those can be categorized into two parts:

The loss of radiance

$$extinction = absorption + out\text{-}scattering$$

The increase of radiance

$$radiance\text{-}incr = emission + in\text{-}scattering$$

Together those result in the final light that arrives at the viewer

$$final\text{-}radiance = radiance - extinction + radiance\text{-}incr$$

Effects due to emission in reality can mostly be neglected [Unk12]. Contrary in games it is useful for artists and for faking multi-scattering. See chapter 2.1.

(a) Absorption (b) Emission (c) Out-Scattering (d) In-Scattering

Figure 2.1: Four Effects of Scattering

2.1 Scattering

Scattering is the "bouncing" of photons on particles. Two different phenomena can be described, **in-scattering** and **out-scattering**. Light that hits particles and bounces

towards the viewer and light that should go to the viewer but getting bounced away by the particles [Wro14; Unk12; Hoo16]. It is not possible to simulate millions of photons, so phase functions are used to determine the amount of radiocity that is scattered. They are similar to bidirectional reflectance distribution functions (BRDFs) of materials for opaque surfaces.

This scattering differs depending on the particles. For simulating scattering by small molecules like air, the *Rayleigh Phase Function*, see equation 2.2, is a close approximation. It is isotropic but wavelength dependent and without absorption. For aerosol or dust, complex *Mie scattering* is used and more physically correct. This model is strongly anisotropic and takes absorption into account. For easier *Mie scattering* calculation the *Henyey-Greenstein Phase Function*, see equation 2.3, or the even faster *Schlick Phase Function*, see equation 2.4, is highly recommended [Yus13; Wro14; Pat13; Jar08].

$$\textbf{Isotropic Phase Function} \quad \rho_I = \frac{1}{4\pi} \tag{2.1}$$

ρ = phase factor

$$\textbf{Rayleigh Phase Function} \quad \rho_R = \frac{3}{16\pi}(1 + \cos^2(\theta)) \tag{2.2}$$

ρ = phase factor, θ = angle

$$\textbf{Henyey-Greenstein Phase Function} \quad \rho_{HG} = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\cos(\theta))^{1.5}} \tag{2.3}$$

ρ = phase factor, $g \in [-1, 1]$ = relative strength of forward and backward scattering, θ = angle

$$\textbf{Schlick Phase Function} \quad \rho_S = \frac{1 - k^2}{4\pi(1 + k\cos(\theta))^2} \tag{2.4}$$

ρ = phase factor, $k \in [-1, 1]$ = preferential scattering direction, θ = angle

Figure 2.2: Equations of different phase functions [Jar08]

Those scattering events take place multiple times in reality. Because the evaluation of any scattering function is expensive, most realtime applications only consider single-

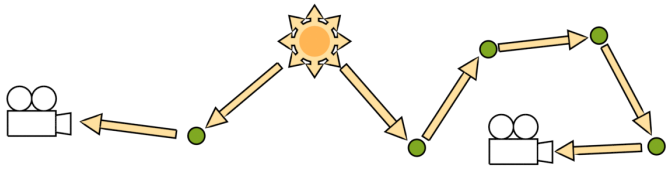


Figure 2.3: Single-scattering (left), multi-scattering (right)

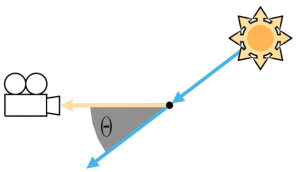


Figure 2.4: Angle θ [Jar08]

scattering. Multi-scattering, the bouncing of one photon on multiple molecules shown in figure 2.3, is often faked with an ambient term equivalent to indirect lighting for global illumination [Pat13].

2.2 Transmittance

Transmittance describes the amount of light that is not diverted or absorbed by any particles and hits the viewer. It is calculated with *Beer-Lambert law*, see equation 2.5. The optical depth for this law is calculated with equation 2.6 in which the extinction is the combination of absorption and out-scattering.

$$\textbf{Beer-Lambert Law} \qquad T = e^{-\tau} \tag{2.5}$$

T = transmittance, τ = optical depth

$$\textbf{Optical depth} \qquad \tau = \sum \mu * \Delta z \tag{2.6}$$

μ = extinction, Δz = interval length, τ = optical depth

3 Approaches

All approaches can be split into three categories: mesh-based, post-processing and 3D volumes. Advantages and disadvantages are discussed in chapter 4.

3.1 Mesh-Based

3.1.1 Billboards

Billboards are semi transparent sprites placed in worldspace and always oriented towards the camera. Often multiple sprites are stacked behind each other. To reduce the effect of rotating sprites, the billboards fade out depending on the distance to the camera.

Figure 3.1: Layered sprites [Che18]

3.1.2 Particles

This approach is similar to the Billboard technique. The sprites are much smaller or only points and managed by the particle system of the game engine. With an advanced particle system this can be very effective.

Figure 3.2: *Unity VFX Graph* [Che18]

3.1.3 Semi Transparent Geometry

Semi transparent 3D geometry is an old approach for creating light shafts. This is done by extruding polygons from the light source and stopping when scene geometry is hit. Alternatively the light geometry can be created manually. If the game engine uses light geometry for shadow casting, this geometry can be directly used for the semi transparent fog geometry.

Figure 3.3: *Zelda: The Wind Walker* [Mit04]

3.2 Post-Processing

3.2.1 Analytical Fog

This approach takes the depth texture of the final image and alpha blends the fog depending on the scene depth. This can be done with any function, most popular are linear, squared or exponential functions. This effect is especially useful to hide unloaded parts of the scene.

Figure 3.4: Depth based fog [Cra14]

3.2.2 Post-Process Ray Marching

Post-process ray marching is a more complex approach for light shafts and is completely done in post-process. In this algorithm rays are sent in screen space from the light source to the current pixel. Those samples are evaluated with the light scattering function and blended onto the final scene image. This can be extended by using an occlusion method for correctly handling scene geometry and generating shadows for the light rays [Mit07].

Figure 3.5: God rays in *Crysis* [che16]

3.3 3D Volumes

3.3.1 Volume Ray Marching

Figure 3.6: Ray marching result [TU09]

The algorithm by Toth et al. performs ray marching from the scene towards the camera. On every ray samples are taken in a defined distance. For each sample the scattering equation is evaluated depending on the last sample while taking shadows into account [TU09].

This can be optimised with *Epipolar Sampling*. By having the center points on the camera and the light, all samples of a ray are on an epipolar line of the camera. After evaluating the scattering equation the result is transformed back into rectangular coordinates. Additionally the use of 1D min/max binary trees for evaluating the shadowmap can increase the performance. Samples at sudden depth value changes (depth breaks) increase accuracy of sharp shadow edges. A detailed explanation is available by Yusov: *"Practical Implementation of Light Scattering Effects Using Epipolar Sampling and 1D Min/Max Binary Trees"* [Yus13].

Another optimising method is *Interleaved Sampling*. Instead of taking a sample at every step, it takes the sample of neighboring pixels [TU09]. With addition of bilateral *Gaussian Blur* and depth-aware up-sampling, the resulting image quality and performance is suitable for games [Gla14].

Guerrilla Games have developed a highly optimized ray marching implementation for clouds [Sch15].

3.3.2 Volumetric Textures

Key idea of this approach is the separation of typical ray marching operations and the storing of intermediate results into 3D textures. This allows a faster and even parallel execution of some steps with compute shaders. Those steps are:

1. Density estimation of participating media
2. Calculating in-scattering light
3. Effect accumulation over depth by ray marching
4. Applying the effect

This requires some type of subdivision of a volume into smaller volumes, called voxelization. Information of every voxel corresponds to one pixel in the 3D texture. When placing the volume containing the voxels aligned with the view frustum, the ray marching operation is a parallel scan through the 3D texture's depth slices [Wro14]. Frustum aligned voxels are called froxels. A detailed description of these steps is part of chapter 6.

4 Comparison

To get an overview which method to choose, we compare them by the following attributes: constraints, performance, visual quality. The evaluation is based on a standard implementation. This does not include for example camera dependent dynamic texture generation for billboards or millions of particles for dense fog. Finally, in table 4.1 those attributes are summarized for all approaches.

4.1 Constraints

Some approaches have limited possibilities. Conditions that can be crucial for determining which approach to choose are described in the following.

4.1.1 Light Sources

Are light dependent interactions, several light sources and different types of lights supported?

- Billboards are light independent.
- Particles have often the option of lit shaders.
- For semi transparent geometry the alpha value can be changed in the shader depending on the distance to the light.
- Analytical fog is light independent.
- Post-process ray marching has the condition to have the light source on screen.
- Volume ray marching needs to be repeated for every light source.
- The volumetric textures approach has the advantage of not repeating all steps for every light. Only step two, calculating in-scattering light, has to loop over all relevant lights.

4.1.2 Scene Scale

Can the approach be up- and down-scaled? For example normal room size compared to landscape with mountains.

- Billboards are good for fog at a far distance, similar to the level of detail (LOD) method.

- Particles are mostly to small for large area dense fog.
- Semi transparent geometry can be scaled like all other geometry.
- Analytical fog is not depending on the scene geometry.
- Post-process ray marching is not depending on the scene geometry.
- Volume ray marching in large areas means more samples or less quality
- Volumetric textures always have the same amount of samples, so fog in the distance is not precise because froxels far away from the camera are larger but still only have one sample.

4.1.3 Dynamic Occlusion

Is it possible to have dynamic objects with shadows or moving lights?

- The sprites of billboards don't cast shadows.
- Most particles engines support shadow casting.
- Semi transparent geometry is the light if extruded from the light source or can normally cast shadows.
- Analytical fog is light independent.
- Post-process ray marching uses screen space occlusion.
- Volume ray marching evaluates the lights and shadows at each sample every frame.
- Volumetric textures evaluate the lights and shadows at each sample every frame as well.

4.1.4 Varying Density

Can the participating media volume be non-uniform? This is normally done by using noise textures to have dense concentration of fog and other places without fog.

- The texture of billboards can be noise textures.
- Particles can be placed non-uniformly. They can be dynamic as well.
- The textures of semi transparent geometry can be noise textures but don't change with the camera view angle.
- Analytical fog can't support varying density because it is calculated in screen space.
- Post-process ray marching can't support varying density, it is calculated in screen space as well.
- Volume ray marching evaluates the density at each sample every frame.
- The volumetric textures approach evaluates the density at each sample every frame as well.

4.1.5 Transparent Objects

Does this approach correctly handle transparent objects in the scene?

- Billboards can be blended in.
- Particles can be blended in.
- Semi transparent geometry is a transparent object itself.
- Analytical fog can be blended in.
- Post-process ray marching is blended in automatically.
- Volume ray marching does not support transparent geometry because the accumulated fog values are only available at the camera and not in the scene.
- The volumetric textures approach does support transparent geometry because the accumulated fog values are available across the scene.

	Billboards	Particles	Semi Transparent Geometry
Light Sources	no	yes	yes
Scene Scale	yes	no	yes
Dynamic Occlusion	no	yes	yes
Varying Density	yes	yes	yes
Transparent Objects	yes	yes	no
Performance	low	medium	low
Visual Quality	medium	medium	low

	Analytical Fog	Post-Process Ray Marching	Volume Ray Marching	Volumetric Textures
Light Sources	no	only on screen	yes	yes
Scene Scale	yes	yes	yes	yes
Dynamic Occlusion	no	yes	yes	yes
Varying Density	no	no	yes	yes
Transparent Objects	yes	yes	no	yes
Performance	low	medium	high	high
Visual Quality	low	high	high	high

Table 4.1: Approach comparison

4.2 Performance

Depending on the approach a different computing power is required. This can be a limiting factor depending on the target platform. The evaluation is based on the core concept of the implementation: loops in shaders, multiple shader passes, complex computations. For a more detailed analysis an implementation and optimization of all approaches with multiple different scenarios would be required. This is out of scope for this thesis.

4.3 Visual Quality

Visual Quality is measured by how close the result of each approach compares to reality. This is very often not required. On one hand a *Low-Poly-Game* does not need accurate light scattering. Semi transparent geometry can even fit better in this case. On the other hand in a photo-realistic first person shooter it would break the immersion if a dusty interior had billboards rotating that don't change with the light. This evaluation is not based on specific tests but on general observations.

Post-process ray marching has very good looking and high quality results but is not accurate compared to reality. The volumetric textures approach is very accurate in regards to single-scattering. This is shown in a comparison between the Frostbite game engine and the physical path tracer *Mitsuba* [Hil15]. Semi transparent geometry has problems with light geometry intersecting normal scene geometry resulting in visible lines [Mit04].

4.4 Choice

What approach to choose is highly dependent on what result and effect is expected. For the god ray effect we decided to implement the volumetric textures approach. It is the newest and most used option in high quality game engines like *Unreal Engine*, *CRYENGINE*, *Frostbite engine*, *Rockstar Advanced Game Engine* and *Unity (High Definition Render Pipeline Package)*. We want to get a deeper understanding what makes this approach superior compared to the others.

5 Programming Environment and External Tools

5.1 Unity

For our implementation we are using Unity version 2019.3.3f1 including the *High Definition Render Pipeline* package version 7.1.8. This allows us to use the core infrastructure of a complete game engine. The *Scriptable Render Pipeline* allows more access to every render step if needed. Visual features including tonemapping, bloom and volumes are disabled.

5.2 Tools

To analyze and debug rendering processes like shaderpasses and compute shader we use *Renderdoc* which can be integrated into Unity. Graphics were made in Unity with use of Sebastian Lague's Debug Viewer [Lag20] and in *Blender*. The demo scenes are created with 3D assets from the *Unity Asset Store*: HQ Autumn Dry Maple Trees by Roman Borisenko [Bor18], Rocky Hills Environment - Light Pack by Tobyfredson [Tob18] and Snaps Prototype | Sci-Fi / Industrial by Asset Store Originals [Ori19].

5.3 Hardware

All tests are performed on NVidea GForce 1060 6GB GPU in combination with an Intel Core i7-7700K CPU and 16GB DIMM-RAM with 3200MHz.

6 Implementation

In this chapter we analyze the Froxel 3D Volume algorithm step by step and present our implementation. Figure 6.1 shows an illustration of scattering of one ray with corresponding equations for calculating the light scattering at a sample point and equation 6.2 for calculating the light accumulation front to back.

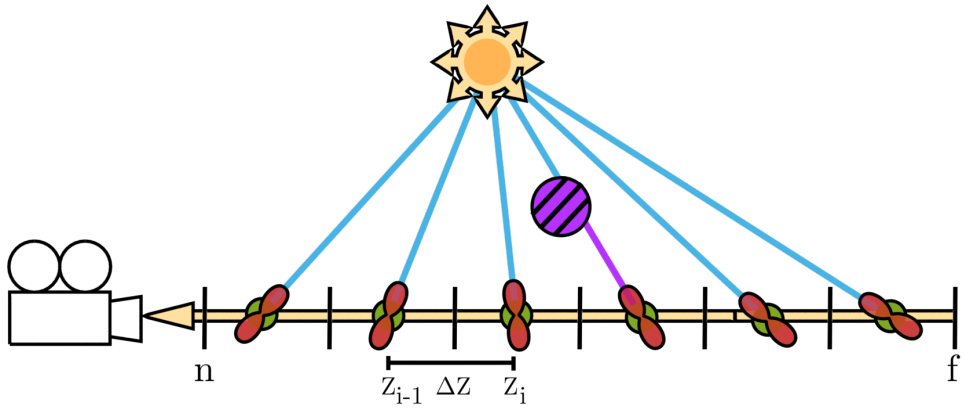
6.1 View Frustum Voxelization

Concept The frustum voxelization is needed for detecting intersections with density volumes, affected light areas and calculating the position of one sample in each froxel. The amount of froxels is defined in each dimension in x - width, y - height and z - depth. Recommended is 160 in width, 90 in height and 64 or 128 in depth dependent on the platform [Wro14]. This results in 921 600 or 1 843 200 froxels. See figure 6.2 for an example.

6.1.1 Generation and Transformation

Concept There are some options to divide the view frustum. One is to divide an unit-cube into voxels and then project them into screenspace with the inverse projection-matrix of the camera. Note that the projection-matrix has a depth factor effect of $1/z$ [Ree15]. This has to be considered for the depth distribution. The positions can also be calculated by using ray vectors pointing from the camera towards the view frustum similar to raycasting. See chapter 7.3 for more information about this option.

Implementation We decided to use simple trilinear interpolation from the view frustum corners. This is slow but possible because we only calculate those positions once at the start of the game and save them relative to the camera. On start we interpolate all (x+1)*(y+1)*(z+1) corners of the froxels in three nested loops, one for each dimension. After that, in another three nested loops, we group together the eight corners of each froxel. The center of each froxel is calculated by adding all eight corners together and dividing the resulting vector by eight. This center will later be our sample point. For sample position variation see chapter 7.3.



$$L_{scat}(x_s) = \sum_{l=0}^{lights} Vis(x_s, l) L(x_s, l) \rho \tag{6.1}$$

x_s = position of sample, l = light index, $Vis(x_s, l)$ = visibility of light at a point, $L(x_s, l)$ = light value from light l at point x_s, ρ = phase function factor

$$L_{accum} = \sum_{z_i=0}^{1} T(z_i) d L_{scat}(z_i) \Delta z (f - n) \tag{6.2}$$

z_i = normalized slice depth, $\Delta z = z_i - z_{i-1}$ = depth of froxel, n = near plane, f = far plane, d = density

Figure 6.1: Implementation graphic based on [Hil15]

All froxels are relative to the camera. To get the worldspace coordinates we multiply them with the local-to-world matrix of the camera. This has to be done every frame. To accelerate those matrix vector multiplications we make use of a compute shader. Getting the result back from the GPU to the CPU takes time and is a bottleneck described in chapter 7.3.1. For a small amount of froxels this is acceptable.

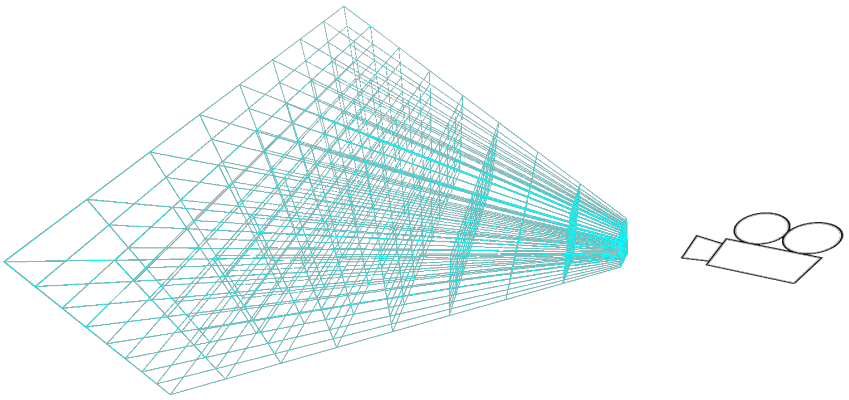


Figure 6.2: Frustum voxelization with 8x8x8 tiles

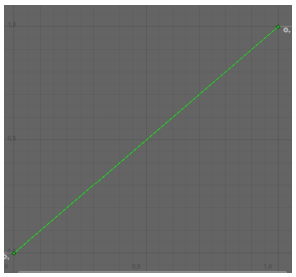
6.1.2 Depth Distribution

Concept Depth distribution defines how the depth slices are distributed with increasing distance to the camera. To distribute the depth slices one could use a fixed function. Generally a distribution with more slices towards the near plane is recommended to reduce aliasing artifacts [Wro14].

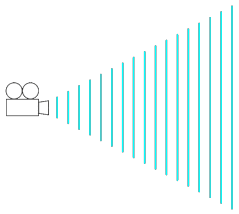
Implementation We use an Unity-Animation-Curve to change and experiment with different values more easily. This curve is defined between 0 and 1 with an output from 0 to 1 and can be directly used in our trilinear interpolation. Later for the step applying effect we need to save the inverted animation curve in a 1D texture because it is not possible to evaluate an animation curve in a shader. See figure 6.3 for the animation curves in respect to the resulting depth distribution with 17 depth slices.

6.2 Density Estimation

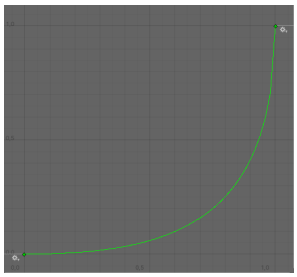
Concept To know how dense the participating media is at a specific position we take samples from the density input. The input includes global fog and local fog volumes consisting of 3D textures in worldspace. This can be done in a compute shader by looping over all volumes, transforming the sample point into the local normalized space of the volume, sampling the 3D texture and adding the global fog value. The result is saved in another 3D texture with the same layout like the froxels. This texture saves the fog color in the R, G and B channel and the extinction factor in the alpha channel.



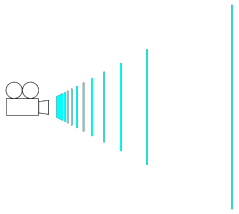
(a) Linear animation curve



(b) Linear depth distribution



(c) Nonlinear animation curve



(d) Nonlinear depth distribution

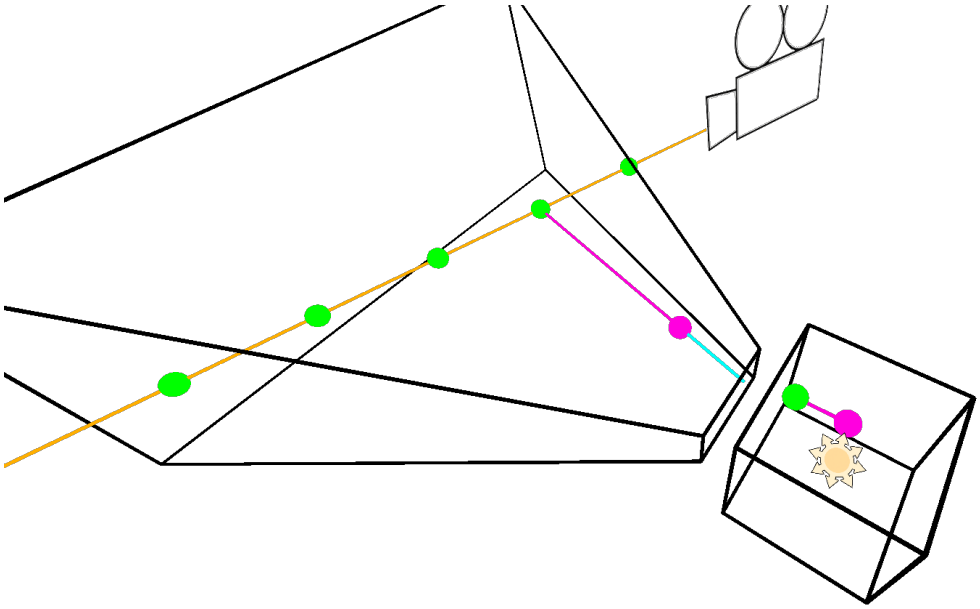
Figure 6.3: Different depth distributions with animation curves

For wavelength dependent extinction, a second 2D texture is necessary [Bau19]. To accelerate this step it is possible to test for intersection of the froxel bounding box with the fog volume bounding box. This can be combined with the detection of samples close to the volume edge to create softer edges [LG18]. See chapter 7.3 for more information.

Implementation Our implementation does not concentrate on this step. We only implement global fog by setting the RGBA values of the resulting 3D texture to a constant.

6.3 Light Calculation

Concept All lights have to inject their information into the froxels. This is done in a compute shader by looping over all lights and calculating for each sample if it is in shadow, what light or shadow color is present and what intensity the light has. This step implements equation 6.1 which includes one of the scattering functions from chapter 2.1. The result is saved in another view frustum aligned 3D texture. Like with density estimation this can be accelerated by testing for intersections between the froxel with bounding boxes of the effective light area.



green = samples, purple = occluding depth point

Figure 6.4: Visibility test of one sample

Implementation Due to limited access to Unity's internal processes, we decided to calculate the visibility of a light manually. This is done by first rendering the depth texture from the light position with a second camera. This light-camera's view frustum covers the same area as the light. Then the position of each sample is projected with the light-camera's view projection matrix into light clip space like shown in figure 6.4. Now we test if the sample is inside the light's view frustum by checking if it is inside the unit-cube. If the sample is inside the light's view frustum we transform the projected position from the [(-1,-1,-1),(1,1,1)] unit-cube to a [(0,0,0),(1,1,1)] unit-cube. Finally we only need to compare the projected position with the value we read from the depth texture with the projected position's x and y coordinates. If 1 minus the depth is smaller than the projected point, the sample is in shadow, else the sample is lit by the light and we can insert the light values into the light 3D texture. See image 6.5a for a buffer slice example.

For our example we use the isotropic phase function shown in equation 2.1.

(a) Light buffer texture slice 40

(b) Accumulated buffer slice 128

(c) Accumulated buffer at index x=35, y=32 from front (left) to back (right)

Figure 6.5: Buffer examples from image 6.6

6.4 Accumulation by Ray Marching

Concept This step is necessary to get the accurate accumulation of the fog values over the depth. Strictly speaking this is not done by ray marching because we have no ray, only a x and y coordinate of the density and 3D light texture. By looping over the z values of the 3D textures from the near plane to the far plane, we can calculate for each z value the accumulated radiance and optical depth. Those values are saved in another 3D texture. See equation 6.2.

Implementation The optical depth of the current froxel is calculated by multiplying the extinction with the slice depth delta. The accumulated optical depth is the sum of the current optical depth and the optical depth of the froxel from the last slice. Because the sample lies between two depth slices, only half of the current optical depth is added to the last result and saved in the accumulation 3D texture's alpha channel. The other half is added after saving the value and the result is stored outside the loop for the next slice.

To calculate the transmittance in between the depth slices and the transmittance from the camera to the depth slice, we use *Beer-Lambert law*, see equation 2.5. The density is simply read from the density 3D texture. The light values are read from the light 3D texture and multiplied with the transmittance from the camera to the depth slice. The final color radiance of the current sample is calculated by multiplying the transmittance of the depth slice with the density color values and the light color values. This is added

to the radiance of the last depth. The result is saved in the accumulation 3D texture. See image 6.5b for an example of the last depth slice and image 6.5c for all values of one x and y coordinate of the accumulation 3D texture across the depth.

6.5 Applying Effect

Concept The final effect is applied as a fullscreen pass. The accumulated light value can be read from the 3D texture depending on the linear depth of the cameras depth texture because the algorithm calculated the values from the near plane towards the far plane. The opacity for alpha blending is calculated from the optical depth in the alpha channel with equation 6.3. It is also possible to implement compatibility with semi-transparent objects because the accumulated values are available before and after any semi-transparent objects. They can simply be blended in [Wro14].

$$\textbf{Opacity} \quad \alpha = 1 - T \tag{6.3}$$

T = transmittance, α = opacity

Implementation To correctly apply our depth distribution from chapter 6.1.2 we have to modify our depth value. First we read the depth value d from the depth buffer and linearize it as d_{lin}. Then we use d_{lin} to sample the inverted animation curve texture so we get d_{curve}. Finally we can use d_{curve} together with the normalized screen coordinates to sample the 3D texture with the accumulated values and use the RGB values as output combined with the calculated opacity from optical depth. Blend mode is one minus source alpha and Z write off.

Transparent objects are ignored in our implementation.

Figure 6.6: Demo result with 64x64x128 froxels

7 Evaluation

7.1 Volumetric Textures for God Rays

If the desired effects are only god rays, the post-process ray marching approach is easier to implement. For overall fog including god rays, the chosen volumetric textures approach offers a unified solution with many possibilities for optimization, see chapter 7.3.

For implementation in a game engine it might be worth testing a combination of several approaches. For example *Rockstar's Rockstar Advanced Game Engine* uses volumetric textures for near fog and volume ray marching for clouds [Bau19].

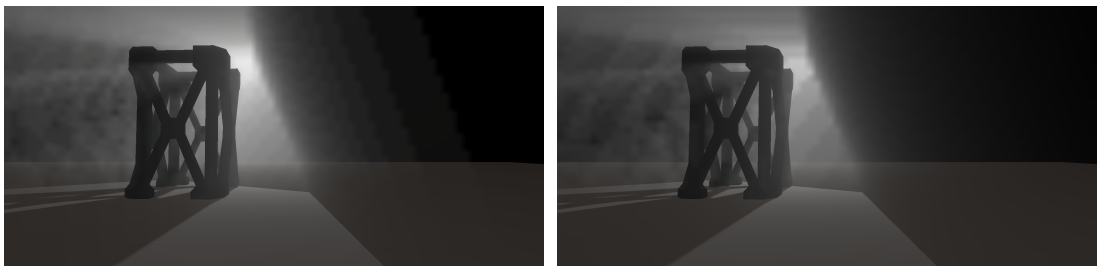
7.2 Depth Distribution Analysis

During testing we discovered that a depth distribution with more slices near the camera (non linear) does not always yield better results. In figure 7.1, 7.2 and 7.3 we compare three different positions of the camera each with linear and non linear depth distribution shown in chapter 6.1.2 in figure 6.3. All pictures have 80 x 45 x 64 froxels.

In figure 7.1 the camera is inside the lit fog volume and looking away from the light. The non linear distribution has far less artifacts. In figure 7.2 the camera is inside the lit fog volume and looking towards the light. Artifacts are visible in both pictures but in different areas. In figure 7.3 the camera is outside the lit fog volume and looking perpendicular to the light. The linear distribution has less artifacts than the non linear distribution.

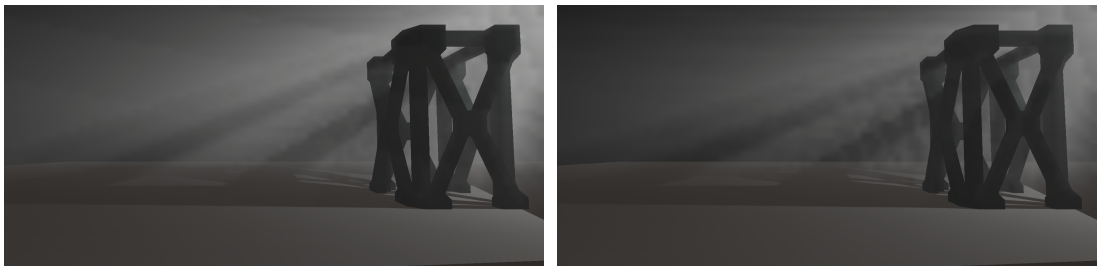
(a) Linear distribution (b) Non linear distribution

Figure 7.1: Camera inside looking away from the light



(a) Linear distribution (b) Non linear distribution

Figure 7.2: Camera inside looking towards the light



(a) Linear distribution (b) Non linear distribution

Figure 7.3: Camera outside looking perpendicular to the light

7.3 Implementation Optimizations

This implementation is not meant to be a finished solution to use in real-time applications. For production ready rendering in games a lot of optimizations are necessary.

Figure 6.6 shows our results without any optimizations mentioned in this chapter.

7.3.1 Performance

Most notable are the "GetBuffer" and "SetBuffer" calls to move data between the GPU and the CPU. This can be avoided by simply staying on the GPU. This could possibly be achieved by global shader variables and buffers. Another option is to calculate the sample point on the fly by calculating them as points on rays from the near plane to the far plane. Each ray would correspond to one x and y coordinate of the 3D textures. Note that the depths are bound to x and y because otherwise all points of the same depth would not lie on a flat plane but on a curved one.

To optimize the renderpipeline we have to consider the injection points of our compute shaders. In our implementation they are currently called in the Unity update loop. Density estimation can be called anytime because it does not depend on other steps. Light calculation requires the geometry and light of the scene so any time after shadow calculations is possible. Accumulation by ray marching has to be done after the two previous steps. Applying effect has to be performed after the previous steps and after transparency.

For many light sources close together light clustering improves performance because step light calculation has to be performed only once per cluster.

7.3.2 Visual

The most needed visual improvement is to reduce the visibility of artifacts especially while the camera is moving. One commonly used approach is to implement temporal integration, also referred to as temporal reprojection. This is a way to use previous frames and combine the calculated values with the current frame [Hil15; LG18]. Other improvements to increase visual quality include down-sampling and blurring shadowmaps [Wro14], jittering samples, applying filters, detecting partial overlapping fog-volumes with voxels and including importance sampling [LG18].

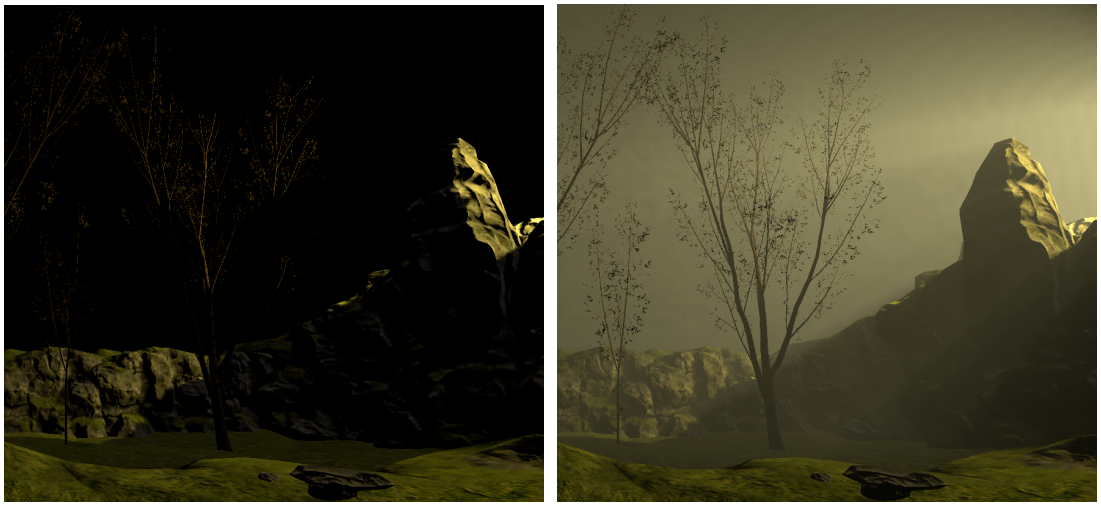
7.3.3 Ideas

For accelerating the calculation of many small lights it could be beneficial to have a hierarchical structure of different sized froxels. This would allow for faster detection of light affected froxels.

Another idea is to change the depth distribution of the depth slices dynamically to allow higher precision where it is needed. Chapter 7.2 shows possible improvements for certain situations.

8 Conclusion

In this thesis, we created the god ray effect by analyzing how light scattering is present in reality, categorized and compared advantages and disadvantages of all commonly used methods of creating this effect and implemented the volumetric textures approach in Unity. This implementation visualizes the underlying concept and is the base for implementing and testing future optimizations.



(a) Scattering effect off (b) Scattering effect on

Figure 8.1: Demo result of our implementation

List of Figures

List of Tables

Glossary

absorption Transformation of a photon's energy into other types of energy like heat.

anisotropic Light intensity depends on direction, opposite of isotropic.

artifact Unwanted sudden color change.

compute shader Special code programmed to run on the GPU. It allows parallel computations following the single instruction, multiple data (SIMD) concept.

emission The emergence of photons by molecules due to various chemical and physical reasons.

epipolar Mathematical concept which describes a relation between two points of view.

god ray Fog with visible light and shadow stripes. Also called light shafts, sunbeams or crepuscular rays.

isotropic Same light intensity regardless of direction, opposite of anisotropic.

optical depth Describes the loss of radiance through a medium.

post-process An image processing step where only the final image and the depth buffer is available.

radiance The amount of radiation defined for an area.

Bibliography

[Bau19] F. Bauer. *Creating the Atmospheric World of Red Dead Redemption 2: A Complete and Integrated Solution*. July 30, 2019. URL: https://advances.realtimerendering.com/s2019/slides_public_release.pptx (visited on 04/05/2020).

[Bor18] R. Borisenko. *HQ Autumn Dry Maple Trees*. Oct. 15, 2018. URL: https://assetstore.unity.com/packages/3d/vegetation/trees/hq-autumn-dry-maple-trees-93117 (visited on 04/12/2020).

[che16] chetanjags. *Volumetric Lighting : SunShafts*. Feb. 2, 2016. URL: https://chetanjags.wordpress.com/2016/02/02/volumetric-lighting-sunshafts/ (visited on 04/05/2020).

[Che18] S. Cherkasov. Oct. 23, 2018. URL: https://www.gamasutra.com/blogs/SvyatoslavCherkasov/20181023/329151/Graveyard_Keeper_How_the_graphics_effects_are_made.php (visited on 04/05/2020).

[Cra14] S. Craitoiu. *Create a fog shader*. July 22, 2014. URL: http://in2gpu.com/2014/07/22/create-fog-shader/ (visited on 04/05/2020).

[Gla14] B. Glatzel. *Volumetric Lighting for Many Lights in Lord of the Fallen*. Digital Dragon Conference. Deck13 Interactive GmbH. 2014. URL: https://www.slideshare.net/BenjaminGlatzel/volumetric-lighting-for-many-lights-in-lords-of-the-fallen (visited on 01/27/2020).

[Hil15] S. Hillaire. *Towards Unified and Physically-Based Volumetric Lighting in Frostbite*. Siggraph 2015. Electronic Arts / Frostbite. 2015. URL: http://advances.realtimerendering.com/s2015/Frostbite%20PB%20and%20unified%20volumetrics.pptx (visited on 02/17/2020).

[Hoo16] N. Hoobler. *Fast, Flexible, Physically-Based Volumetric Light Scattering*. GDC 2016. NVIDEA Developer Technology. Mar. 16, 2016. URL: https://developer.nvidia.com/sites/default/files/akamai/gameworks/downloads/papers/NVVL/Fast_Flexible_Physically-Based_Volumetric_Light_Scattering.pdf (visited on 01/27/2020).

[Jar08] W. Jarosz. "Efficient Monte Carlo Methods for Light Transport in Scattering Media." PhD thesis. UC San Diego, Sept. 2008. Chap. 4.

[Lag20] S. Lague. *Debug Viewer*. Mar. 3, 2020. URL: https://github.com/SebLague/DebugViewer (visited on 04/12/2020).

[LG18] S. Lagarde and E. Golubev. *The Road toward Unified Rendering with Unity's High Definition Render Pipeline*. page 135 - 186. Unity Technologies. Aug. 25, 2018. URL: http://advances.realtimerendering.com/s2018/Siggraph%202018%20HDRP%20talk_with%20notes.pdf (visited on 01/29/2020).

[Mit04] J. Mitchell. *Light Shafts*. GDC 2004. 2004. URL: http://developer.amd.com/wordpress/media/2012/10/Mitchell_LightShafts.pdf (visited on 01/27/2020).

[Mit07] K. Mitchell. *Volumetric Light Scattering as a Post-Process*. GPU Gems 3. Electronic Arts / NVIDEA. Aug. 12, 2007. URL: https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-13-volumetric-light-scattering-post-process (visited on 02/18/2020).

[Ori19] A. S. Originals. *Snaps Prototype | Sci-Fi / Industrial*. Dec. 23, 2019. URL: https://assetstore.unity.com/packages/3d/environments/sci-fi/snaps-prototype-sci-fi-industrial-136759 (visited on 04/12/2020).

[Pat13] Patapom. *Real-Time Volumetric Rendering - What's a participating medium?* 2013. URL: https://patapom.com/topics/Revision2013/Revision%202013%20-%20Real-time%20Volumetric%20Rendering%20Course%20Notes.pdf (visited on 03/26/2020).

[Ree15] N. Reed. *Depth Precision Visualized*. July 15, 2015. URL: https://developer.nvidia.com/content/depth-precision-visualized (visited on 03/27/2020).

[Sch15] A. Schneider. *The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn*. Siggraph 2015. Guerrilla Games. Aug. 26, 2015. URL: http://advances.realtimerendering.com/s2015/The%20Real-time%20Volumetric%20Cloudscapes%20of%20Horizon%20-%20Zero%20Dawn%20-%20ARTR.pdf (visited on 04/07/2020).

[Tob18] Tobyfredson. *Rocky Hills Environment - Light Pack*. Feb. 28, 2018. URL: https://assetstore.unity.com/packages/3d/environments/landscapes/rocky-hills-environment-light-pack-89939 (visited on 04/12/2020).

[TU09] B. Toth and T. Umenhoffer. "Real-time Volumetric Lighting in Participating Media." In: *Eurographics 2009 - Short Papers*. Ed. by P. Alliez and M. Magnor. The Eurographics Association, 2009. DOI: 10.2312/egs.20091048.

[Unk12] Unknown. *IVB Atmospheric Light Scattering*. Intel Software - Game Dev. Aug. 6, 2012. URL: https://software.intel.com/en-us/articles/ivb-atmospheric-light-scattering (visited on 01/27/2020).

[Unk17] Unknown. *low angle photography of crepuscular rays in forest wallpaper*. Dec. 30, 2017. URL: https://www.wallpaperflare.com/low-angle-photography-of-crepuscular-rays-in-forest-wallpaper-21336/800x600 (visited on 04/12/2020).

[Wro14] B. Wronski. *Volumetric Fog: Unified compute shader based solution to atmospheric scattering*. Siggraph2014. Ubisoft Montreal. 2014. URL: https://bartwronski.files.wordpress.com/2014/08/bwronski_volumetric_fog_siggraph2014.pdf (visited on 01/27/2020).

[Yus13] E. Yusov. *Practical Implementation of Light Scattering Effects Using Epipolar Sampling and 1D Min/Max Binary Trees*. GDC 2013. Intel Corporation. Mar. 27, 2013. URL: https://software.intel.com/sites/default/files/managed/b9/1d/gdc2013-lightscattering-final.pdf (visited on 01/27/2020).