Evaluation on Technologies for Real-time Volumetric Cloud Self-occlusion Rendering

Jing Wang, BAI

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

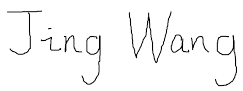
(Augmented and Virtual Reality)

Supervisor: Michael Manzke

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.



Jing Wang

August 17, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Jing Wang

August 17, 2022

Evaluation on Technologies for Real-time Volumetric Cloud Self-occlusion Rendering

Jing Wang, Master of Science in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Michael Manzke

Clouds are a beautiful phenomenon in nature. The real-time volumetric cloud rendering technologies are already widely used in recent video games. To render photorealistic clouds, we should research and reproduce the properties of the clouds. Among these properties, cloud self-shadowing or self-occlusion is an important cue to the perception of cloud shape. In this study, I investigate four main real-time approaches to render the self-shadow of clouds: i) Secondary ray marching; ii) Exponential shadow map; iii) Beer shadow map; iv) Fourier opacity map. Their memory footprint and render time are tested under different settings and situations. According to the results, secondary ray marching almost does not consume extra memory and is able to render fairly realistic cloud self-shadow, while it needs a long time for rendering when taking many samples. On the other hand, the other three methods store occlusion information in shadow maps. Though shadow maps consume some memory, they can accelerate rendering when at high screen resolution or when lots of cloud need to be rendered in a scene. However, their visual results are not as accurate as secondary ray marching. Based on the evaluation on the four approaches, the artists can choose a suitable one according to their purposes.

Acknowledgments

A sincere thank to my supervisor: Michael Manzke. He supports and guides me throughout the whole project. He provides me with a lot of advice on how to write a good dissertation.

I also want to thank my family. I would not be able to go on this journey in Ireland without their emotional support.

Thanks, should also go to my friends: Tobias Alexander Franke, TJ McGrath Daly, Yvain Li, Ziqi Luo, Xiangpeng Fu. It is my luck to meet like-minded folks at such a perfect moment.

Jing Wang

University of Dublin, Trinity College
August 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Clouds are beautiful and usually affect people's mood. Nowadays, lots of video games and films are using clouds to make specific atmospheres for their scenarios. For example, *Life of Pi* uses thick and dark clouds to express the character's nervousness, while *Red Dead Redemption 2* foils the lonely and peaceful mood with sunset and yellowish clouds. These years, with the rapid evolution of hardware, especially graphics processing unit (GPU), the demand on the real-time cloudscape rendering technology is increasingly getting higher.

Figure 1.1: Cloud scene in films and games.The left image is from *Life of Pi* Lee (2012), the right image is from *Red Dead Redemption 2* Rockstar (2018).

Regarding cloudscape real-time rendering in game industry, many explorations have been done in the last several decades. The main representations of cloud are five:

Polygon: Cloud's polygon can be made by fluid simulation and its lighting information can be precomputed for rendering. Whereas, polygon cloud tends to have monotonous shapes that wispy cloud is difficult to be made with fluid simulation. In addition, cloud animation on polygon cloud is usually costly.

Billboard: Billboard cloud represents clouds with 2D images. This approach is less costly but the artists need to prepare a large amount of different appearances of

various orientations for the changing time and weather. Meanwhile, it is difficult to reproduce self-occlusion of clouds with only 2D images.

Sky dome: Sky dome cloud precomputes several cloud sets to make sky dome. The sky dome can be blended into the atmosphere while it is not suitable to represent the cloud evolve over time.

Particle: This approach represents clouds as volumes of particles. The particle representation can render clouds efficiently with precomputed dynamic lighting from the sun and atmosphere. Whereas the particle cloud is mostly limited to cumulus-like shapes.

Volumetric cloud: Volumetric cloud uses voxel to represent clouds. This approach is promising in the evolution of cloud, self-shadowing etc. However, volumetric cloud has many problems, for example, it usually needs lots of 3D textures which requires large memory, while it renders clouds using ray marching which is costly technique for real-time applications.

Among these approaches, even though volumetric cloud is severely expensive in render time, the pleasing visual results and the flexibility of cloud animation is attracting more and more people to optimise it and introduce it to real-time games. In 2015, Schneider and Vos (2015) proposed a series of techniques to render volumetric cloud in about 2 milliseconds that made volumetric cloud affordable in real-time games. Following this breakthrough, game engine engineers gradually introduced volumetric cloud to their sky systems Hillaire (2016, 2020); Bauer (2019).

On the other hand, real-time volumetric cloud technology still has a few limitations. For instance, clouds have high albedo (close to 1) which means most light enters the cloud will be scattered. The scattered light enters another part of the cloud and the scattering process happens again. Even though one scattering process is simple, when the scattering happens hundreds of thousands of times inside a volume of cloud, tracing all the rays to render realistic appearance is not affordable in real-time Hillaire (2020). Due to this, only single scattering is computed in real-time games by far. On top of that, how to draw a large scale of cloud, how to simulate the cloud motion etc. are also open questions now. Hufnagel and Held (2012) and Schneider (2017) concludes some future works for cloudscape real-time rendering:

1. Heterogeneous cloud scene rendering. The constituents of clouds changes over altitude. Usually, the cloud at low altitude consists of large water droplets, while the constituent changes to ice particles at high altitude. The phase functions of these

constituent are usually not homogeneous. This requires specific cloud representations and lighting models for different clouds.

2. Large-scale cloud scenes. Nowadays, we can repeat weather map, which is a 2D texture describing the weather information, especially the distribution of cloud, to render broad cloudscape. When it comes to a large scene, e.g., a cloud scene over the whole Europe, the repeated pattern will break the immersiveness.

3. Cloud-to-cloud shadows and inter-reflection. Since cloud is transparent, light may travel through clouds, this needs a different model to represent this property than those of opaque objects. On top of this, the in-scattering effect is also difficult to reproduce because the heavy computation.

4. The inclusion of atmospheric scattering models for lighting. The illumination from the atmospheric also affects the appearance of clouds. For instance, the clouds under sunny weather are usually brighter than rainy weather.

5. Cloud at different scales. When looking at the clouds from close and far, they should show different levels of details (LOD). Especially, how to implement continuous LOD should be considered in real-time applications.

6. Temporal cloud animations. Emulating the motion of clouds by translating a 2D image of cloud distribution is relatively easy, while the more distortion behaviours, such as the stormy cloud rotating like a tornado, can hardly be simulated by simple manipulation on such a 2D image.

7. Interaction between clouds and other objects. When we are standing on the ground, the clouds are distant, we do not interact with them. However, when we drive a flight and dive into the clouds or we have a especially tall mountain in the scene, the interaction between clouds and other objects should be considered.

Among these future works, this study mainly focus on the solutions for cloud occlusion.

The cloud occlusion has three main effects. First, cloud will produce self-shadowing. When the cloud is thick, less light can penetrate through the clouds. As a result, the bottom of the cloud is usually darker than the top. In addition, if we move above the cloud layer and overlook the clouds, the taller clouds will cast shadows onto the lower clouds. Second, cloud will cast shadow on the ground. The clouds in the sky occludes the sunlight, so we can observe the cloud shadow on the ground. Third, cloud will produce light shaft. Light shaft is also known as "god ray". Compared with the 2D shadow on the surface of objects, the light shaft is 3D shadow. Under specific weather condition,

the shined air will get brighter due to Tyndall effect. This phenomenon often appears after rain. Since the clouds occludes the sunlight, the occluded space beneath the clouds receives less light and appears darker. The main interest of this study is rendering the cloud itself, therefore, the technologies for cloud self-shadowing is investigated.

By far, there are mainly four methods are used for rendering clouds self-shadows. Schneider and Vos (2015) used secondary ray marching which is a classic method. This method can render fairly realistic appearance while it is expensive in render time as the number of samples increases. Bauer (2019) used ESM (exponential shadow map Annen et al. (2008)) to store the occlusion information and render cloud shadows. Hillaire (2020) proposed BSM (Beer shadow map) to improve the visual results of Bauer (2019). Besides, even not directly applied on clouds, Hillaire (2015) used FOM (Fourier opacity map Jansen and Bavoil (2010)) for rendering shadows from fogs and many other participating medias. The latter three methods are using the ideas of shadow mapping to record the occlusion due to clouds. This idea tends to consume less time in rendering, while the visual results are usually not as good as secondary ray marching. Moreover, the additional shadow maps consume some memory. Even though people have proposed many methods for cloud occlusion, there is almost no research has compared and evaluated the performance of them. For example, even the secondary ray marching is slow, when the scene has only a few clouds, it is might be preferable to use secondary ray marching than other methods which are faster but less visually pleasing. On top of that, the needs of resources to create shadow maps varies among these shadow mapping methods, e.g. the memory consumption is different. Considering these facts, we need a guideline to tell people which method is more suitable for their purposes. In this study, experiments are designed to test the performance, including the memory footprint, rendering time etc, of cloud occlusion methods. In a nutshell, the contribution of this study is I evaluated four main methods for cloud self-occlusion, and gave a guidance of how to choose a proper methods according to different demands.

In this paper, volumetric cloud rendering technology is investigated. In Chapter 2, the history and recent technologies for cloudscape rendering is reviewed. In Chapter 3, the design of experiments and the implementation is described. Later, the experiment results are discussed in Chapter 4. Lastly, a conclusion and future works can be found in the last Chapter 5.

Chapter 2

State of the Art

In this chapter, the fundamental of cloud formation and volume rendering will be introduced. Then the recent technologies used for describing occlusion from clouds are reviewed.

2.1 Cloud in Nature

Clouds in nature can be divided into many species. According to their features, e.g. shape and altitude, the most common clouds are 3 groups: strato clouds, alto clouds and cirro clouds Schneider and Vos (2015). Figure 2.1 is an illustration of these clouds. Strato clouds are low-level clouds. They exist between about 0 to 2 km above sea level. Stratus, cumulus and stratocumulus belong to strato clouds. Alto clouds are medium-level clouds. They exist between about 2 to 8 km above sea level. Altostratus and altocumulus belong to alto clouds. Cirro clouds are high-level clouds. They exist between about 3 to 18 km above sea level. Cirrostratus, cirrus and cirrocumulus belong to cirro clouds.

The formation of cloud is already well investigated. Clouds consist of vapours and small ice particles. The higher altitude the clouds at, the less vapours they have. In addition, the temperature at high altitude is low, hence the vapours will change to small ice particles. Therefore, the high-level clouds are usually smaller and less thick. Clausse and Facy (1961) indicates some useful empirical results for simulation cloud formation. For example, vapours rise with the heat from earth and the wind direction varies over altitude. These observations and derived rules are greatly helpful to build a cloud system for real-time rendering.

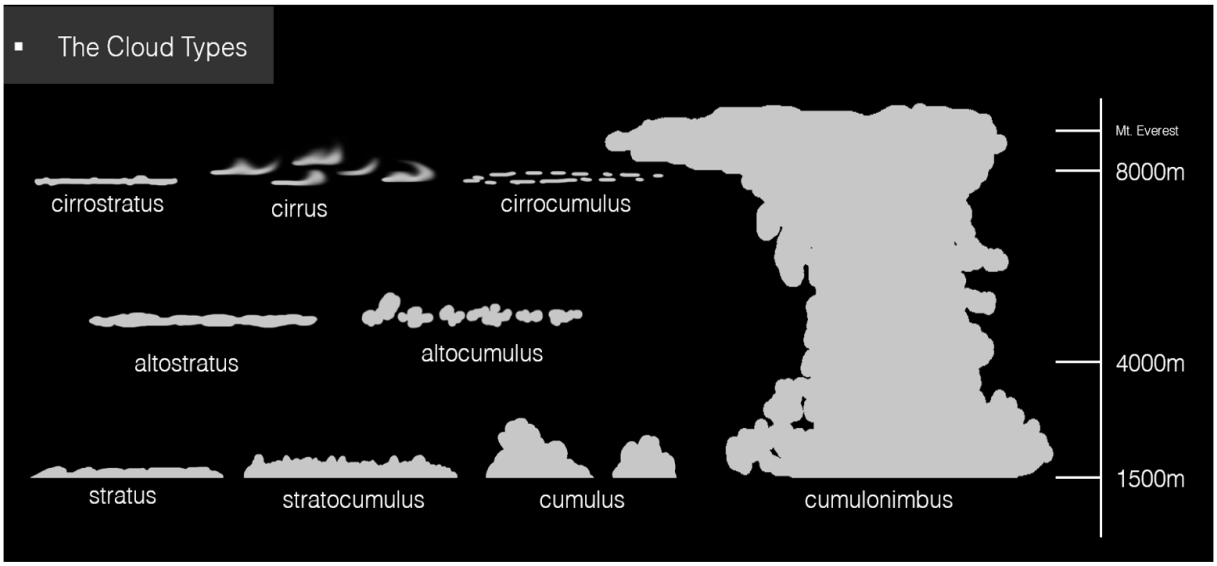


Figure 2.1: Illustration of main cloud types Schneider and Vos (2015)

2.2 Volume Rendering

The theory of volume rendering is already widely used in rendering clouds. Cloud is a type of participating media. To render participating media, four volume scattering processes should be considered: absorption, emission, out-scattering and in-scattering Pharr et al. (2016). An illustration of these processes is presented in Figure 2.2.

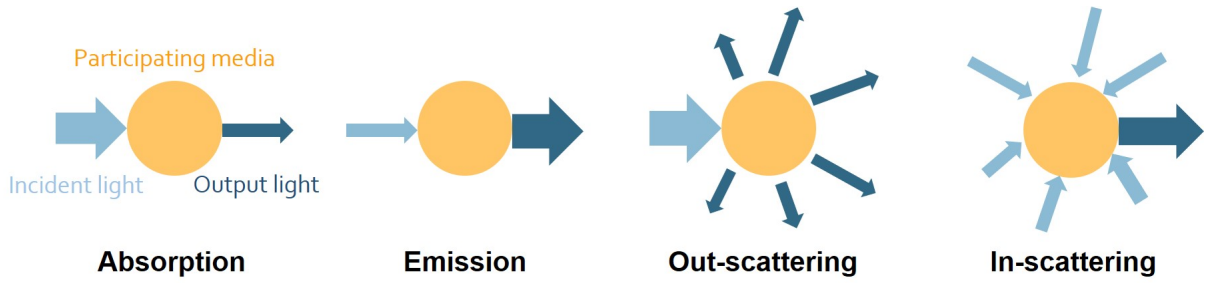


Figure 2.2: Illustration of four volume scattering processes.

Absorption denotes that light will be absorbed when travelling through participating media. For example, when light passes fog and cloud, some radiance will be absorbed. The effect of medium's absorption can be described by the cross section and the absorption coefficient σ_a as:

$$L_o(p,\omega) - L_i(p,-\omega) = \mathrm{d}L_o(p,\omega) = -\sigma_a(p,\omega)L_i(p,-\omega)\,\mathrm{d}t \tag{2.1}$$

Here, L_i and L_o are input light and output light, p is position, ω is the direction of light, σ_a is absorption coefficient, t is step distance. Beer's law Beer (1852) indicates the trans-

mittance reduces exponentially with distance. Based on Beer's law, the transmittance T_r with the effect of absorption after travelling d distance is:

$$T_r(p' \to p) = e^{-\int_0^d \sigma_a(p+t\omega,\omega)\,\mathrm{d}t} \tag{2.2}$$

Typically, T_r ranges from 0 to 1 denoting the proportion of light travelled through participating media.

While absorption leads to reduction of radiance as the ray passes through the participating medium, emission will increase the radiance. Some participating media, e.g. fire, will emit radiance. The differential equation of the radiance due to emission can be described as:

$$\mathrm{d}L_o(p,\omega) = L_e(p,\omega)\,\mathrm{d}t \tag{2.3}$$

Here L_e is the light emit per unit distance.

The last basic light interaction happens in participating media is scattering. Scattering process has two main effects, out-scattering and in-scattering. Our-scattering deflects light to different directions which reduces the radiance exiting along a direction. The probability of out-scattering can be described by the scattering coefficient σ_s. Similar to absorption, the reduction of radiance along direction ω due to out-scattering is given by:

$$L_o(p,\omega) - L_i(p,-\omega) = \mathrm{d}L_o(p,\omega) = -\sigma_s(p,\omega)L_i(p,-\omega)\,\mathrm{d}t \tag{2.4}$$

Accounting both absorption and out-scattering together as attenuation, the attenuation coefficient $\sigma_t(p,\omega) = \sigma_a(p,\omega) + \sigma_s(p,\omega)$. Therefore, the final transmittance can be expressed as:

$$T_r(p' \to p) = e^{-\int_0^d \sigma_t(p+t\omega,\omega)\,\mathrm{d}t} \tag{2.5}$$

In-scattering is an effect of the light deflected by nearby media enters current media unit and increases the amount of radiance. The total added radiance per unit distance due to in-scattering is given by:

$$\mathrm{d}L_o(p,\omega) = L_s(p,\omega)\,\mathrm{d}t \tag{2.6}$$

$$L_s(p,\omega) = \sigma_s(p,\omega)\int_{S^2} P(p,\omega_i,\omega)L_i(p,\omega_i)\,\mathrm{d}\omega_i \tag{2.7}$$

$$\int_{S^2} P(p,\omega_i,\omega)\,\mathrm{d}\omega_i = 1 \tag{2.8}$$

Here P is phase function describing how much light is scattered from ω_i to ω. A tricky thing in the equations is that scattering happens multiple times in a participating media. To render it correct, we need to trace a tremendous amount of the scattering rays which is not affordable in real-time rendering. As a result, by far only single scattering is computed in most real-time applications, while the multiple can be approximated by Wrenninge et al. (2013).

Finally, based on the theory above, the final radiative transfer equation can be expressed as:

$$L_i(p,\omega) = T_r(p_0 \to p)L_o(p_0,-\omega) + \int_0^t T_r(p' \to p)L_s(p',-\omega)\,\mathrm{d}t' \tag{2.9}$$

2.3 Offline Participating Media Rendering Technology

This section reviews how people render participating media offline. So far, the offline technologies for rendering participating media mainly have 5 types of methods Wu et al. (2022): volumetric density estimation, virtual ray light, point, Monte Carlo and neural networks.

Volumetric density estimation based technologies. Typically, this approach includes two passes Jensen and Christensen (1998); Jarosz et al. (2011): i) lighting pass; ii) rendering pass. The basic idea of lighting pass is to distribute light photons or rays, beams from the light source to the scene and store them into the medium element, and then estimate their density in the medium. While the rendering pass is to compute the contribution of to the camera rays based on the density and the property of the media to render the scene. Křivánek et al. (2014) indicates the different representations are suitable for simulating the light transport in different participating media. For example, light photon is suitable for high-order scattering media, while beam is suitable for low-order scattering media. According to this, since cloud is high-order scattering participating media, light photon should be a good choice.

Virtual point/ray/beam light based technologies are similar to volumetric density estimation based technologies. Similarly, this type of method distribute photons or rays or beams into the scene, whereas the way they use the cached light distribution is different. VPLs (virtual point lights) Keller (1997); Arbree et al. (2008) assumes every photon is a point light and compute the illumination. Later, improved methods, VRLs (virtual ray lights) Novák et al. (2012b) and VBLs (virtual beam lights) Novák et al. (2012a) are proposed for faster rendering and the capability in wider materials. These methods

are faster than those based on volumetric density estimation, while they have singularity issues that the potential tiny distance between photons may yield artefacts.

Point based technologies also have lighting pass and rendering pass, whereas it makes use of the geometry of the cached point cloud in its lighting pass. Wang et al. Wang et al. (2016) introduces point-based global illuminaiton Christensen (2008) to participating media rendering. They organised the volume and surface in hierarchies and computed single, double and multiple scattering separately. Later, Liang et al. Liang et al. (2019) introduced frequency analysis theory for single scattering computation. This improvement results higher quality while using fewer volume samples. After making use of the geometry information, this group of algorithms is able to produce high-quality results in a shorter time than volumetric density estimation based technologies. Meanwhile, compared with virtual point/ray/beam light based technologies, one advantage of this type of methods is the capability to compute single scattering. However, the limitation is that the methods are assumed on homogeneous media such as uniform fog, while the extension to heterogeneous media still requires further works.

Monte Carlo solutions are unbiased and simple, so they have been widely used in participating media rendering. Monte Carlo based methods have distance sampling operation and phase function sampling operation for path tracing integration Kajiya and Von Herzen (1984). The distance sampling usually consider the attenuation over the distance the light travels. However, high-order scattering media may cause millions of scattered lights in the medium which needs a long time to render. As for the phase function, the highly anisotropic media requires sampling of a high-frequency phase function. High-frequency phase function sampling is difficult to converge, otherwise, the result tends to be noisy. Lafortune and Willems Lafortune and Willems (1996) improved the convergence rate by extending bidirectional path tracing. Further, new technologies were proposed or introduced to improve the Monte Carlo based methods, e.g., next event estimation Jakob and Marschner (2012); Weber et al. (2017), path guiding Herholz et al. (2019); Deng et al. (2020) and zero-variance random walks Křivánek and d'Eon (2014).

Recently, neural networks are also introduced to participating media rendering. For example, regarding cloudscape rendering, Kallweit et al.Kallweit et al. (2017) proposed to use a radiance-predicting neural networks. They store the shading configuration in the radiance-predicting neural networks which should include location, the light source the density structure of the volumetric cloud and so on. During rendering, multiple scattering is computed by the neural network, while Monte Carlo is used for single scattering.

2.4 Real-time Participating Media Rendering Technology

This section reviews two main type of participating media in real-time video games: cloud and fog.

For a long time, under the limitation of hardware, people tend to use billboard cloud in real-time applications. To make the motion of billborad more realistic, Guerette et al. Guerrette (2014) proposed a method to give a illusion of moving cloud with visual flow technique. However, billboard cloud does not involve the evolution of cloud shape over time, and it also does not interact with the weather.

Some works proposed to render clouds as volumes of particles. Harris and Lastra (2002) renders a group of particles in the distance to approximate clouds. Since the distant clouds have less details, we can represent the cloud with less particles and update it in a low frame rate. Later, Yusov (2014) introduced pre-integrated lighting which takes the dynamic lighting of the sun in to account. Meanwhile, Yusov made use of rasterizer ordered views to avoid particle-like look. Even the particle representation can render visual pleasing clouds, it is limited to cumulus, while struggles to represent status and cumulonimbus.

These years, regarding the atmosphere in real time games, the participating media in a distance, such as cloud in the sky, and the one close to the player, such fog, are rendered in two different ways.

Schneider and Vos (2015) proposed a cloud system which renders volumetric cloud in real-time games. Volumetric cloud uses voxel to represent the density of clouds. To create cloud-like shapes, the artist can pre-generated 3D Perlin Noise and Worley Noise at different frequencies and mix them together to control the cloud shapes. Ray marching is used for rendering clouds. However, computing multiple scattering in real-time is not feasible, so only single scattering is considered for every ray. With suitable noise and ray marching, we can already form cloud-like shapes and produce cloud-like appearance, but a frame may take over 20 millisecond to render, which is still not affordable for real-time games. If we consider real-time as 60 frame per second, one frame should be processed within 16.7 millisecond. Therefore, Schneider and Vos (2015) optimised the algorithm by temporal integration. They only render 1 pixel in a square of 16 pixels in one frame, and render the 16 pixels in order. With this optimisation and several other trade-off, they squeeze the render time to approximately 2 millisecond. Thanks to this algorithm, more and more games can build better sky systems with dynamic clouds Hillaire (2016); Rockstar (2018).

Compared with cloud, fog is usually close to the players and player can walk into an

foggy area. Due to this, rendering fog needs more details and it usually needs to take the nearby light sources into account. Historically, the same approaches for clouds, such as billboard and ray marching, are also used for fog. However, similarly, these methods are either not dynamic enough or very slow, people want to find a better way to render fog. Bartlomiej Wronski (2014) proposed a new method to render volumetric fog. Bartlomiej introduces Light Propagation Volumes Kaplanyan (2009) to rendering participating media rendering and proposed a new technology to render the volumetric fog in real-time. He first created a 3D texture aligned with camera frustum, and then store the illumination information into the froxels. Finally, he used ray-marching to render the result of the atmospheric effects on screen. With some optimisation technologies, this technologies could be run in real-time, while it can only be used for near area. As for clouds, which is usually several thousands metre away from the viewer, it is to expensive to build a huge 3D texture to store the illumination of a broad space. On the other hand, it is possible to combine this technology with volumetric cloud rendering. For instance, Rockstar (2018) uses the volumetric fog technology in near area to render fog, while uses ray marching for distant clouds.

2.5 Noise for Modeling Clouds

Noise is a good tool for procedural content generation. The shapes of fluid and terrains, even animal communities distribution can be approximated by proper noises. When modeling clouds, Perlin Noise Perlin (2002) and Worley Noise Worley (1996) are widely used Schneider and Vos (2015).

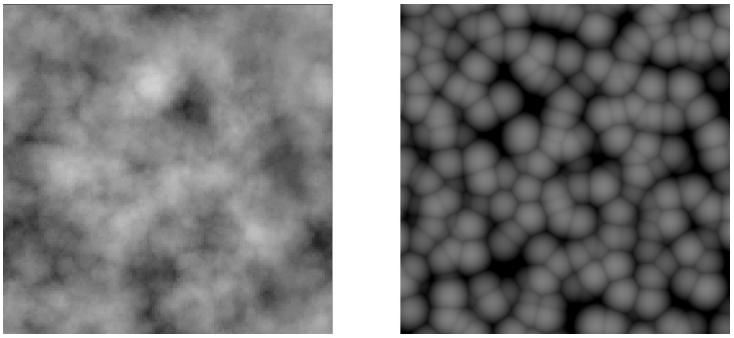


Figure 2.3: Examples of Perlin Noise (left) and Worley Noise (right).

Perlin Noise can be used for any sort of wave-like, undulating material or texture. For example, it could be used for procedural terrain, fire effects, water etc. Compared with totally random noise, Perlin noise is continuous, so it can represent continuous change of shapes. Typically, 3D Perlin Noise is used for giving a basic shape for volumetric clouds.

An example of 2D Perlin Noise is shown in Figure 2.3. When generate Perlin Noise, for every unit cube, we should assign a random vector to every vertex. Then, to compute the values of the voxels, we first compute the vectors from the 8 vertices to the current voxel, and then compute the dot product of this vector and the vector assigned to the corresponding vertex. After getting the 8 dot product values, interpolate them using cubic interpolation to get a value for the current voxel. Repeating this process for all the voxels, we will get a 3D Perlin Noise. Generally, Perlin Noise with only one frequency is not a sufficient approximation to the complex and subtle shapes of those nature objects, e.g. mountain and clouds. Therefore, FBM (Fractal Brownian Motion) Decreusefond and Üstünel (1998) is used for adding more details. FBM is to mix Perlin Noise of different frequencies together, while higher frequency has lower amplitude. The Perlin Noise can create the basic round shapes of clouds, while FBM approach makes some nice whispy shapes.

However, many types of clouds also have cauliflower shapes, Perlin noise alone cannot cover this feature. To solve this problem, Worley Noise is combined to Perlin Noise. Worley noise, sometimes called cellular noise, is a distance value pattern. It is widely used for simulating textures of water, stone and creature skin. An example of 2D Worley Noise is shown in Figure 2.3. To generate a 3D Worley Noise, we first need to distribute several points inside the space. For every voxel, we compute its shortest distance to the nearest point and store this value. After computing this value for every voxel repeatedly, we get a 3D Worley Noise. On top of this, we can optimise this generation process by dividing the space into small cells and only compute the distance with points in current or adjacent cells. By using Worley Noise of different frequencies, we are able to generate billowy shapes for clouds. A comparison of using Perlin Noise and Worley Noise is displayed in Figure 2.4.

2.6 Ray Marching and Cloud Occlusion

Ray marching is a typical method for rendering volumetric in real-time applications. This section reviews the techniques for computing cloud occlusion in ray marching.

Ray marching casts rays (view ray) from the camera through the pixel on the view plane to the world and compute the pixel value according to the intersections with objects. Regarding volumetric cloud, for every intersection with the cloud, we will first query the density from 3D textures, and then compute the lighting with regard to the density occlusion and many other information, lastly added the contribution of this intersection to the pixel. The radiative transfer equation is described as Equation 2.9. In cloud rendering, this equation can be divided into the background term L_B and the cloud term

Figure 2.4: Cloud modeling with Perlin and Worley Noise Schneider and Vos (2015). The left image is using Perlin Noise with low frequency Worley Noise, while the right image is after adding high frequency Worley Noise.

L_C:

$$\begin{aligned} L_i(p,\omega) &= T_r(p_0 \to p)L_o(p_0,-\omega) + \int_0^t T_r(p' \to p)L_s(p',-\omega)\,\mathrm{d}t' \\ &= L_B + L_C \qquad (2.10) \\ L_B &= T_r(p_0 \to p)L_o(p_0,-\omega) \qquad (2.11) \\ L_C &= \int_0^t T_r(p' \to p)L_s(p',-\omega)\,\mathrm{d}t' \qquad (2.12) \end{aligned}$$

The main purpose of ray marching is to solve L_C. Since the analytical answer of L_C almost cannot be computed, in ray marching, we use a discretised form instead:

$$\begin{aligned} L_C &= \int_0^t T_r(p' \to p)L_s(p',-\omega)\,\mathrm{d}t' \qquad (2.13) \\ &= \sum_{i=0}^{n} T_r((p+id) \to p)L_s(p',-\omega)len(d) \qquad (2.14) \end{aligned}$$

Here d denotes the vector of the view ray. $len(d)$ is the length of vector d. Meanwhile, the transmittance can be approximated by Beer's law:

$$T_r((p+id) \to p) = \prod_{j=0}^{i} \exp(-Den(p+jd)C_{den}len(d)) \qquad (2.15)$$

In this equation, Den is the density of the cloud, C_{den} is the coefficient to describe the absorption and scattering property of the media. Based on this theory, we can approach L_C, whereas how much radiance is scattered towards the camera at the sample point (L_S) in Equation 2.14 is another problem needs to be solved. Since multiple scattering is too complex to be solved within several millisecond, usually only single scattering is computed in real-time applications, while the multiple scattering effect can be approximated by If we only take single scattering into account, L_S is:

$$L_s(p, -\omega) = \sum_{i}^{N_{light}} P(p, \omega_i, \omega) V(p, \omega_i) L_i \tag{2.16}$$

P is phase function, V is visibility term telling how much light is occluded, L_i is the radiance from the light source, N_{light} is the number of light sources. The phase function P is same for most clouds and can be approximated by Henyey-Greenstein function Henyey and Greenstein (1941), the radiance cast by the light source L_i is easy to know, hence, the main task turns to computing the occlusion V. Typically, people are using four methods to compute occlusion.

Secondary ray marching Schneider and Vos (2015) is a classic way for solving occlusion. For every sample along the view ray (referred as view ray sample), this method casts another ray (light ray) towards the light source. Along the light ray, it takes samples (referred as light ray sample) to compute how much occlusion is between the view ray sample point and the light source. If detected cloud along the light ray, then use Beer's Law (Equation 2.15) to compute attenuation to get how much radiance finally arrives the view ray sample point.

The other three method are shadow mapping methods. Before casting view rays, they first cast rays from the view of the light source along the light direction. Then, for every ray, they compute and store the occlusion information in different ways in their shadow maps. When to solve Equation 2.16, they directly query the generated shadow maps for occlusion information. Exponential shadow map (ESM) Bauer (2019), Beer shadow map (BSM) Hillaire (2020), Fourier opacity map (FOM)Hillaire (2016) are three typical ways for this purpose.

ESM stores the depth until the first occlusion in its shadow map. After detecting occlusion, it compute an exponential value f:

$$f = \exp(cz) \tag{2.17}$$

Here c is a parameter to control the speed of attenuation, z is the distance from light

source to the first occlusion. During rendering, ESM computes occlusion as:

$$\begin{aligned} V &= f * \exp(-cd) && (2.18)\\ &= \exp(cz) * \exp(-cd) && (2.19)\\ &= \exp(-c(d-z)) && (2.20) \end{aligned}$$

Here d is the distance from sample point to the light source. According to this function, as d increases, V decreases to 0. When V is larger than 1, it will be clamped to 1. On top of this, the shadow map can be pre-filtered to generate soft shadow.

BSM is an improved ESM designed for rendering clouds. BSM assumes only a volume of clouds exists along a ray. To approximate the occlusion, BSM stores front depth, mean density, max optical distance, 3 values in the shadow maps. Front depth is similar to ESM, recording the distance between the first occlusion and the light source. Mean density, or mean attenuation coefficient, is the average density of cloud along the ray. Max optical distance records the maximum optical depth which is the maximum occlusion from a volume of cloud. If we get a longer optical distance, it will be clamped to the maximum optical depth because the assumption. When rendering, the occlusion is computed as:

$$\begin{aligned} D &= \min(D_{max}, \sigma_{mean} \max(0, (d-z))) && (2.21)\\ V &= \exp(-D) && (2.22) \end{aligned}$$

D is optical depth, D_{max} is max optical depth, σ_{mean} is mean density, d is the distance from the sample point to the light source, z is front depth.

FOM approximates the distribution of occlusion with Fourier series, and store the coefficients of Fourier series in the shadow maps. FOM express the transmittance d away from the light source as:

$$\begin{aligned} T_r(d) &= e^{-\int_0^d \sigma(t)\,\mathrm{d}t} && (2.23)\\ \sigma_t &\approx \frac{a_0}{2} + \sum_{k=1}^{n} a_k \cos(2\pi k z) + \sum_{k=1}^{n} b_k \sin(2\pi k z) && (2.24)\\ a_k &= -2 \sum_i \ln(1-\alpha_i) \cos(2\pi k d_i) && (2.25)\\ b_k &= -2 \sum_i \ln(1-\alpha_i) \sin(2\pi k d_i) && (2.26) \end{aligned}$$

Here d is a normalised distance from the light source (ranges from 0 to 1), a_k and b_k are Fourier series coefficients, α_i is density. Generally, more coefficients can better approximate the occlusion distribution, while it consumes more memory. As a trade-off, 7

coefficients, a_0 to a_3 and b_1 to b_3, are enough for most cases.

Chapter 3

Design and Implementation

In this chapter, the purpose together with the details of the experiments will be described in the beginning. Following the design of the experiments, the implementation is described in the next Section 3.2.

3.1 Experiment Design

3.1.1 Purpose

So far, people have been using various techniques to render clouds self-shadowing. However, only a few researches investigated the performance of these techniques and most of them only slightly mention the difference of methods. For example, Hillaire (2020) indicates BSM consumes less memory than FOM, whereas they did not explicitly present how much memory BSM saves than FOM. We do not have a guide to tell us when to use which method. Therefore, this study designs experiments to compare four popular methods for clouds self-shadowing. As explained in previous chapter, the four methods are:

1. Secondary Ray Marching
2. Exponential Shadow Map (BSM)
3. Beer Shadow Map (BSM)
4. Fourier Opacity Map (FOM)

The last three can be included as shadow mapping methods. In the experiments, we investigate the memory consumption of different methods, the render time with different settings and in different situations.

3.1.2 Experiment1: memory footprint

Secondary ray marching does not require a shadow map and any other precomputed resource, so it almost does not consume extra memory. Regarding the rest three methods, they store the occlusion information with different forms and the memory footprints are various. This experiment tests and compares the memory consumption of different methods. Regarding the shadow mapping methods, the memory consumption of shadow map at different resolutions (512×512 and 1024×1024) will be computed.

3.1.3 Experiment3: render time of shadow mapping methods at different shadow map resolution

As for shadow mapping methods, when using high resolution shadow maps, they can get more accurate occlusion information due to higher sampling rate. However, more pixels means they should spend longer time to generate shadow maps. To investigate the relationship between render time and shadow map resolution, this experiment will measure the total render time and shadow map generation time of ESM, BSM and FOM at 256×256, 512×512 and 1024×1024.

3.1.4 Experiment3: render time with different number of samples

Generally, the more samples we have, the more correct results we get. On the other hand, sampling too many times slows rendering. In real-time rendering, we need to find a proper balance between visual result and render time. According to the resource we can use, the proper number of samples will change. In this experiment, the total render time and the shadow map generation time are to be compared. For secondary ray marching, the render time is tested with different numbers of samples, from 1 sample to 50 samples, along the light ray. For ESM, BSM and FOM, the render time is tested with different numbers of samples, from 10 to 300, when generating shadow maps.

3.1.5 Experiment4: render time with different resolutions

As for ray marching, larger resolution means a need of casting more rays through the pixels to render clouds. The total number of samples we take along the view rays is described as Equation 3.1. N_{view} is the number of sample for all the view rays, W and H are the width and height of the resolution, $\bar{C}$ is the average ratio of clouds on the screen,

$\bar{S}$ is the average number of samples we take for every view ray.

$$N_{view} = W * H * \bar{C} * \bar{S} \tag{3.1}$$

For every sample on the view rays, we also need to take more samples to compute the occlusion. In secondary ray marching, we can set how many samples (S_L) we take along the light ray, while we only need to query once from the already computed shadow map when using shadow map methods. The number of samples along the light ray is described as Equation 3.2:

$$N_{light} = \begin{cases} S_L, & \text{if secondary ray marching.} \\ 1, & \text{if shadow mapping methods.} \end{cases} \tag{3.2}$$

$$\tag{3.3}$$

Regarding the number of samples when generating shadow maps, this can be described as Equation 3.4. N_{SM} is the number of samples for shadow maps, W_{SW} and H_{SM} are the width and height of the resolution of shadow map, S_S is the number of samples we take along the light direction for computing the occlusion.

$$N_{SM} = W_{SM} * H_{SM} * S_S \tag{3.4}$$

Even though the process to get one sample is not identical when using different methods, the total number of samples could be an indication of the complexity. Based on this theory, the total number of samples N can be described by:

$$N = N_{view} * N_{light} + N_{SM} \tag{3.5}$$

Since cloud is transparent object, it does not require a very precise geometry information. Therefore, W and H are usually larger than W_{SW} and H_{SW}. On top of this, secondary ray marching has larger N_{light}, so larger resolution should slow secondary ray marching more than the shadow mapping methods. In experiment 3, the effect of resolution on the rendering time will be investigated. The performance of these methods on 720p (1280 x 720), 1080p (1440 x 1080), 2k (2560 x 1440) will be tested.

3.1.6 Experiment5: render time with different cloud coverage rates

In different applications, the requirement for rendering cloudscape is different. Even in the same application but different scene, the coverage of cloud also varies. Hence, the performance of these methods with different cloud coverage rate is also worth investigating. According to Equation 3.1 to 3.5, the cloud coverage $\bar{C}$ also affects all the four methods. Similar to the theory described in Section 3.1.5, larger $\bar{C}$ should have move impact on secondary ray marching than shadow mapping methods. Therefore, this experiment tests the performance of these method with different cloud coverage rates, from 10 % to 100 %.

3.1.7 Common Settings

All the experiments are run in one computer. The CPU used is i7-11850H, the GPU used is RTX-3080, the total available memory is 32 GB. All the programme is written in OpenGL and C++. Meanwhile, to simulate the cases in real-time applications, temporal integration of the scattered light solution is added to accelerate rendering Schneider and Vos (2015). Specifically, only one fourth of the total pixels are rendered in one frame. Every four pixels are rendered in order, therefore, every four frames is a completed full-size frame.

3.2 Implementation

3.2.1 Prerequisite

The four cloud seld-occlusion methods have some common settings and requirements, these common parts will be described in this section.

Perlin-Worley noise and Worley noise are used for modeling clouds. Two 3D textures are used. The first 3D texture has four channels, the R channel stores Perlin-Worley noise, while G, R, A channels store Worley noise of higher frequency. This 3D texture is to form the basic cloud shapes. The second texture has three channels, these channel store increasingly higher frequency Worley noise. This texture is to further erode the cloud to make wispy shape.

One weather map is used to control the cloud. The weather map has three channels. The R channel stores the coverage of clouds. Higher R value means more cloud exists in the area. The G channel stores the precipitation rate of clouds. Higher G value means the cloud includes more vapour, so the cloud absorb more light and the cloud looks

darker. The B channel stores the height of the clouds. 0 denotes to stratus, 0.5 denotes to stratocumulus, 1 denotes cumulus.

The scene is set to on a sphere, so the cloud will fall in the horizon. The height of the ground is set to 200,000 metre. The bottom of the cloud layer is at 201,000 metre, while the top of the cloud layer is at 204,000 metre.

Ray marching is implemented for volumentric cloud rendering. The illuminance on the cloud has two parts, single scattering and an ambient light to approximate multiple scattering. The computation of the ambient light is refering to . Meanwhile, the background, in other words, the sky excludes the clouds, is also rendered with the method proposed by Preetham et al. (1999). As for the details of the ray marching and the four cloud self-occlusion methods, their implementations are described in the following sections.

3.2.2 Ray Marching

The pseudocode for ray marching is shown in Algorithm 1. The get_light() function is the key term that investigated in this study. The four cloud self-occlusion methods are using different ways to approach the correct result for this function.

Algorithm 1 Pseudocode for Ray Marching

Variables
p: current position
dir: normalised ray direction vector
σ_s: scattering rate
num_step: number of maximum steps
$step\_dist$: distance of one step
L: the amount of light arrives at current point
$final\_light$: final light
$light\_dir$: normalised direction vector to light source
$perlin\_worley$: 3D texture of Perlin-worley noise
$step$: current step
$density$: density of cloud
T: total transmittance
$phase$: result of phase function

Functions
texture(): sample value from a texture
get_light(): a function to compute how much light arrives
HG(): Henyey-Greenstein phase function

Procedure
$T = 1$;
$p = camera\_pos$;
$step = 0$;
$final\_light = 0$;

while $step < num\_step$ **do**
    $p = p + dir * step\_dist$;
    $density = \text{texture}(perlin\_worley, p)$;
    $T \mathrel{*}= \exp(-density * step\_dist)$;
    $L = \text{get\_light}(p)$;
    $phase = \text{HG}(dir, light\_dir)$;
    $final\_light \mathrel{+}= L * density * \sigma_s * phase * T * step\_dist$;
    $step{+}{+}$;
end while

return $final\_light$, T;

3.2.3 Secondary Ray Marching

Algorithm 2 is how secondary ray marching is implemented for the experiments.

Algorithm 2 Pseudocode for Secondary Ray Marching

Variables
p: current position
num_light_step: number of maximum steps
$step\_dist$: distance of one step
L: the amount of radiance arrives current point
$light\_dir$: normalised direction vector to light source
$perlin\_worley$: 3D texture of Perlin-worley noise
$step$: current step
$density$: density of cloud
T: total transmittance
$light\_source\_intensity$: luminance of light source

Functions
texture(): sample value from a texture

15

16 ***Procedure***

17 $T = 1$;

18 $p = start\_pos$;

19 $step = 0$;

20

21 **while** $step < num\_light\_step$ **do**

22 $p = p + light\_dir * step\_dist$;

23 $density = \text{texture}(perlin\_worley, p)$;

24 $T \mathrel{*}= \exp(-density * step\_dist)$;

25 $step{+}{+}$;

26 **end while**

27

28 **return** $T * light\_source\_intensity$;

3.2.4 Exponential Shadow Map

Algorithm 3 is how ESM is generated for the experiments.

Algorithm 3 Pseudocode for ESM Generation

1 ***Variables***

2 p: current position

3 num_light_step: number of maximum steps

4 $step\_dist$: distance of one step

5 L: the amount of radiance arrives current point

6 $light\_dir$: normalised direction vector to light source

7 $perlin\_worley$: 3D texture of Perlin-worley noise

8 $step$: current step

9 $density$: density of cloud

10 T: total transmittance

11 $bound\_texture$: shadow map texture

12 $pixel$: the pixel to process

13 c: a parameter to control the attenuation speed

14 MAX_DEPTH: maximum depth

15 $THRESHOLD$: threshold to decide the depth from the light source

16

17 ***Functions***

18 texture(): sample value from a texture

19 get_world_pos(): compute world space position from normalised device coordinates position
20
21 ***Procedure***
22 **for** *pixel* in *bound_texture* **do**
23 $world\_pos$ = get_world_pos(pixel);
24 $p = world\_pos$;
25 $T = 1$;
26 $step = 0$;
27 **while** $step < num\_light\_step$ **do**
28 $p = p + light\_dir * step\_dist$;
29 $density$ = texture($perlin\_worley$, p);
30 T *= $\exp(-density * step\_distance)$;
31 $depth$ += $step\_dist$;
32 $step$++;
33 **if** $T < THRESHOLD$ **then** break;
34 **end if**
35 **end while**
36 $bound\_texture[pixel] = \exp(c * depth/MAX\_DEPTH)$;
37 **end for**
38
39 **return** *bound_texture*;

After computing ESM, ESM is used to compute how much radiance arrives the sample point. Algorithm 4 shows how ESM is used.

Algorithm 4 Pseudocode for Computing Illuminance with ESM

1 ***Variables***
2 p: current position
3 *texcoords*: texture coordinates
4 *occlusion_depth*: depth of first occlusion
5 *normalised_depth*: normalised depth
6 T: total transmittance
7 *light_source_intensity*: luminance of light source
8
9 ***Functions***
10 texture(): sample value from a texture
11 get_texcoords(): get the corresponding coordinate on the shadow map texture

get_depth(): get the depth from light source perspective

Procedure

$texcoords$ = get_texcoords(p);
$occlusion\_depth$ = texture(ESM, $texcoords$);
$normalised\_depth$ = get_depth(p) / MAX_DEPTH;
T = clamp(exp($-c * normalised\_depth$) $* occlusion\_depth$);

return T * $light\_source\_intensity$;

3.2.5 Beer Shadow Map

Algorithm 5 is how BSM is generated for the experiments.

Algorithm 5 Pseudocode for BSM Generation

Variables

p: current position
num_light_step: number of maximum steps
$step\_dist$: distance of one step
$light\_dir$: normalised direction vector to light source
$perlin\_worley$: 3D texture of Perlin-worley noise
$step$: current step
$density$: density of cloud
$front\_depth$: the start depth of occlusion
num_sample: the number of total samples
$sum\_density$: accumulated density
$mean\_density$: average density
$max\_optical\_depth$: maximum optical depth
T: total transmittance
$bound\_texture$: shadow map texture
MAX_DEPTH: maximum depth
$THRESHOLD$: threshold to decide the depth from the light source

Functions

texture(): sample value from a texture
get_world_pos(): compute world space position from normalised device coordinates position

```
23 Procedure
24 for pixel in bound_texture do
25     T = 1;
26     step = 0;
27     front_depth = -1;
28     num_sample = 0;
29     sum_density = 0;
30     max_optical_depth = 0;
31     world_pos = get_world_pos(pixel);
32     p = world_pos;
33     while step < num_light_step do
34         p = p + light_dir * step_dist;
35         density = texture(perlin_worley, p);
36         T *= exp(−density ∗ step_distance)
37         depth += step_distance
38         max_optical_depth += density * step_distance;
39         sum_density += density;
40         num_sample += 1;
41         if T < THRESHOLD and front_depth < 0 then
42             front_depth = depth;
43         end if
44         step++;
45     end while
46     if num_sample > 0 then
47         mean_density = sum_density / num_sample;
48         bound_texture[pixel].r = front_depth;
49         bound_texture[pixel].g = mean_density;
50         bound_texture[pixel].b = max_optical_depth;
51     else
52         bound_texture[pixel].r = MAX_DEPTH;
53         bound_texture[pixel].g = 0;
54         bound_texture[pixel].b = 0;
55     end if
56 end for
57 return bound_texture;
```

After computing BSM, BSM is used to compute how much radiance arrives the sample point. Algorithm 6 shows how BSM is used.

Algorithm 6 Pseudocode for Computing Illuminance with BSM

Variables
p: current position
T: total transmittance
$front\_depth$: the start depth of occlusion
$mean\_density$: average density
$max\_optical\_depth$: maximum optical depth
$light\_source\_intensity$: luminance of light source

Functions
texture(): sample value from a texture
get_texcoords(): get the corresponding coordinate on the shadow map texture
get_depth(): get the depth from light source perspective

Procedure
$texcoords$ = get_texcoords(p);
BSM_info = texture(BSM, $texcoords$);
$front\_depth = BSM\_info$.r;
$mean\_density = BSM\_info$.g;
$max\_optical\_depth = BSM\_info$.b;
$depth$ = get_depth(p);
$optical\_depth$ = min($max\_optical\_depth$, $mean\_depth$ * max(0, $depth$ - $front\_depth$));
T = clamp(exp($-optical\_depth$) $*$ $optical\_depth$);

return T * $light\_source\_intensity$;

3.2.6 Fourier Opacity Map

Algorithm 7 is how FOM is generated for the experiments.

Algorithm 7 Pseudocode for FOM Generation

Variables
p: current position
num_light_step: number of maximum steps
$step\_dist$: distance of one step
$light\_dir$: normalised direction vector to light source
$perlin\_worley$: 3D texture of Perlin-worley noise
$step$: current step

density: density of cloud
a_i, b_i: Fourier series coefficients

Functions
texture(): sample value from a texture
get_world_pos(): compute world space position from normalised device coordinates position

Procedure

for *pixel* in *bound_texture* **do**
    $T = 1$;
    $step = 0$;
    $final\_luminance = 0$;
    $front\_depth$ = -1;
    $a_0 = 0$;
    $a_1 = 0$;
    $a_2 = 0$;
    $a_3 = 0$;
    $b_1 = 0$;
    $b_2 = 0$;
    $b_3 = 0$;
    $world\_pos$ = get_world_pos(pixel);
    $p = world\_pos$
    while $step < num\_light\_step$ **do**
        $p = p$ + $step$ * $light\_dir$;
        $density$ = texture($perlin\_worley$, p);
        $normalised\_depth = step * light\_dir \ / \ MAX\_DEPTH$;
        a_0 += -2 * log(1 - $density$) * cos($2\pi * 0 * normalised\_depth$);
        a_1 += -2 * log(1 - $density$) * cos($2\pi * 1 * normalised\_depth$);
        a_2 += -2 * log(1 - $density$) * cos($2\pi * 2 * normalised\_depth$);
        a_3 += -2 * log(1 - $density$) * cos($2\pi * 3 * normalised\_depth$);
        b_1 += -2 * log(1 - $density$) * sin($2\pi * 1 * normalised\_depth$);
        b_2 += -2 * log(1 - $density$) * sin($2\pi * 2 * normalised\_depth$);
        b_3 += -2 * log(1 - $density$) * sin($2\pi * 3 * normalised\_depth$);
        $step$++;
    end while

44 $bound\_texture1[pixel].r = a_0$;
45 $bound\_texture1[pixel].g = a_1$;
46 $bound\_texture1[pixel].b = a_2$;
47 $bound\_texture1[pixel].a = a_3$;
48 $bound\_texture2[pixel].r = b_1$;
49 $bound\_texture2[pixel].g = b_2$;
50 $bound\_texture2[pixel].b = b_3$;
51 **end for**
52 **return** *bound_texture*;

After computing FOM, FOM is used to compute how much radiance arrives the sample point. Algorithm 8 shows how FOM is used.

Algorithm 8 Pseudocode for Computing Illuminance with FOM

1 ***Variables***
2 p: current position
3 T: total transmittance
4 a_i, b_i: Fourier series coefficients
5 *optical_depth*: optical depth
6 *bound_texture*: shadow map texture
7 *light_source_intensity*: luminance of light source
8
9 ***Functions***
10 texture(): sample value from a texture
11 get_texcoords(): get the corresponding coordinate on the shadow map texture
12 get_depth(): get the depth from light source perspective
13
14 ***Procedure***
15 $texcoords$ = get_texcoords(p);
16 FOM_info1 = texture($bound\_texture1$, $texcoords$);
17 FOM_info2 = texture($bound\_texture2$, $texcoords$);
18 $a_0 = FOM\_info1$.r;
19 $a_1 = FOM\_info1$.g;
20 $a_2 = FOM\_info1$.b;
21 $a_3 = FOM\_info1$.a;
22 $b_1 = FOM\_info2$.r;
23 $b_2 = FOM\_info2$.g;
24 $b_3 = FOM\_info2$.b;

25 $depth = \text{get\_depth}(p)$;

26 $optical\_depth = a_0/2*depth + a_1/(2*\pi*1)*\sin(2*\pi*1*depth) + a_2/(2*\pi*2)*\sin(2*\pi*2*depth) + a_3/(2*\pi*3)*\sin(2*\pi*3*depth) + b_1/(2*\pi*1)*(1-\cos(2*\pi*1*depth)) + b_1/(2*\pi*2)*(1-\cos(2*\pi*2*depth)) + b_1/(2*\pi*3)*(1-\cos(2*\pi*3*depth))$;

27 $T = \text{clamp}(\exp(-optical\_depth))$;

28

29 **return** T * $light\_source\_intensity$;

Chapter 4

Evaluation

4.1 Results

In this section, the results of the experiments will be presented with graphs and tables and these information will be explained later.

4.1.1 Memory footprint

The memory footprints of different occlusion methods are shown in Table 4.1. Secondary ray marching does not require additional texture, so it does not consume extra memory. On the other hand, the shadow mapping methods first render shadow maps before ray marching, so the additional shadow maps takes some memory. ESM stores one value per pixel with 16 bit float, so when it generates 512 $\times$ 512 shadow map, the extra texture takes approximately 4 MB. When using a higher resolution shadow map, it uses a lot more memory, 12 MB at 1024 $\times$ 1024. Regarding BSM and FOM, they store more values in their shadow maps, hence, their memory footprints are higher. Especially, when FOM generate 1024 $\times$ 1024 shadow maps, it needs about 112 MB to store the information, which is a large memory consumption.

4.1.2 Render time with different shadow map resolutions

The render time of shadow mapping methods with different shadow map resolutions are shown in Figure 4.1. When the resolution is low, the difference of the render time among the three methods are smaller than using higher resolution. When at 256 $\times$ 256 resolution, ESM and BSM uses almost similar time to render (about 4.2 milliseconds), FOM takes approximately 0.3 more milliseconds. While in 1024 $\times$ 1024 resolution, ESM consumes about 5.9 milliseconds which is shortest, BSM and FOM use about 0.7 and 2.2

Table 4.1: Methods' memory footprint.

Methods	**Memory Footprint (MB)**
Secondary Ray Marching	0
ESM (512 × 512)	4
ESM (1024 × 1024)	16
BSM (512 × 512)	12
BSM (1024 × 1024)	48
FOM (512 × 512)	28
FOM (1024 × 1024)	112

more milliseconds than ESM respectively.

Figure 4.1 describes the render time of shadow map generation with different shadow map resolutions. The shadow map generation time shows similar behaviours to the total render time. With the increment of shadow map resolution, the generation time of FOM increases fastest among the three methods. ESM consumes about 0.4 milliseconds, BSM consumes about 0.55 milliseconds, FOM consumes about 0.75 milliseconds, to generate 256 × 256 shadow map. When generating 1024 × 1024 shadow map, ESM consumes about 1.6 milliseconds, BSM consumes about 2.5 milliseconds, FOM consumes about 3.7 milliseconds. From 256 × 256 resolution to 1024 × 1024 resolution, ESM uses approximately extra 1.2 milliseconds, BSM uses approximately extra 1.95 milliseconds, FOM uses approximately extra 2.95 milliseconds.

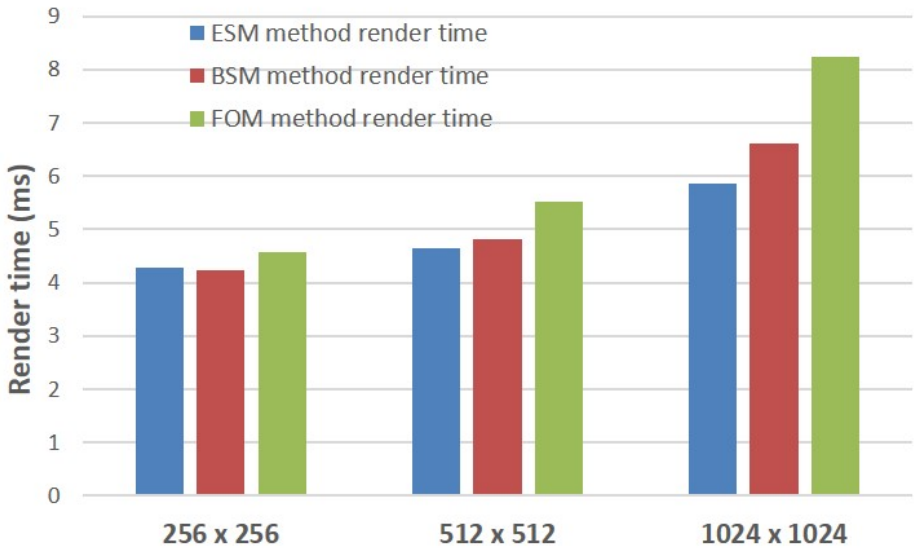


Figure 4.1: Render time of shadow mapping methods with different shadow map resolutions.

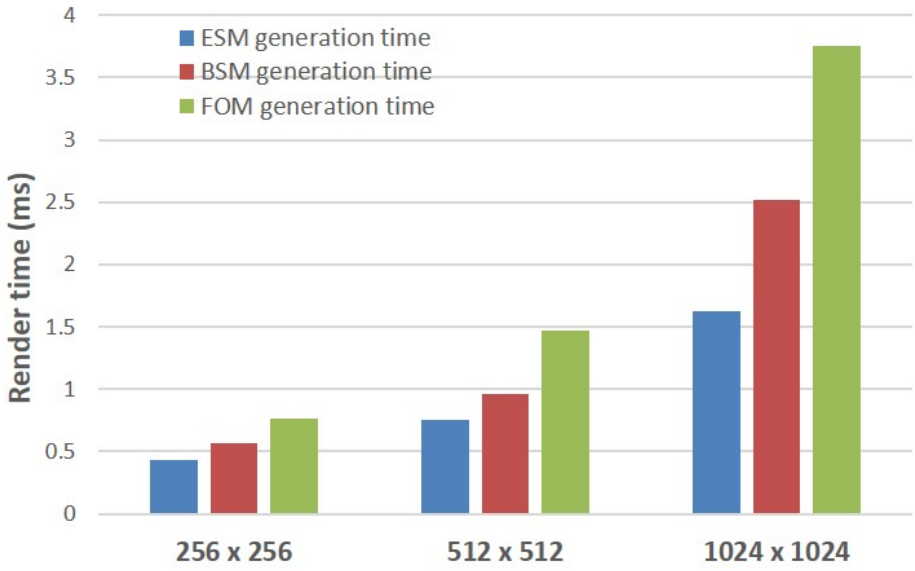


Figure 4.2: Render time of shadow map generation with different shadow map resolutions.

4.1.3 Render time with different number of samples

Since the ways to sample are different in secondary ray marching and the other three shadow mapping methods, the results are divided into two parts.

Secondary ray marching method

The render time of secondary ray marching with different number of samples is shown in Figure 4.3. With the number of samples increasing, the render time increases linearly from about 4 milliseconds when only take one sample to about 9.4 milliseconds after taking 50 samples.

Shadow mapping methods

The render time of three shadow mapping methods and their time for generating shadow maps with different number of samples is shown in Figure 4.4. Among the three methods, FOM takes longest time to generate shadow maps and render clouds, while BSM consumes longer time than ESM on both shadow map generation and rendering in most cases, except when only take a few samples (less than 50), BSM renders clouds faster than ESM.

Summary of methods with typical number of samples

A summary of the render time of methods with some typical parameters are concluded in Table 4.2. Even taking only 10 steps, secondary ray marching is slower than ESM and

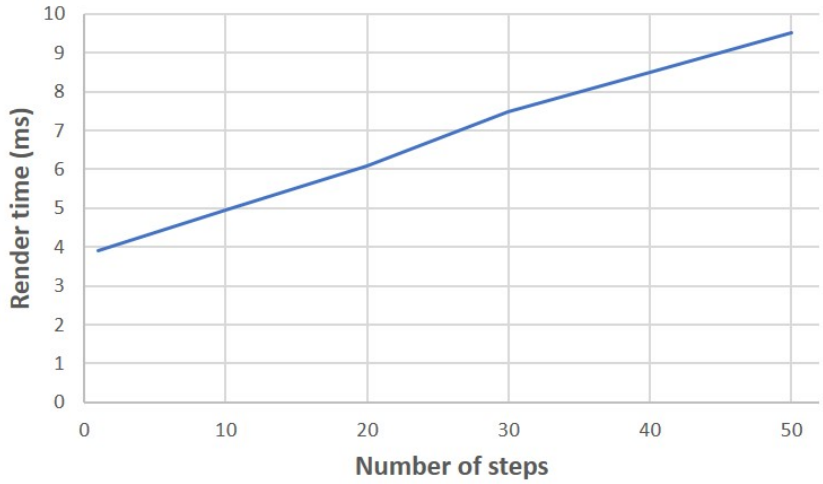


Figure 4.3: Render time of secondary ray marching method with different number of samples.

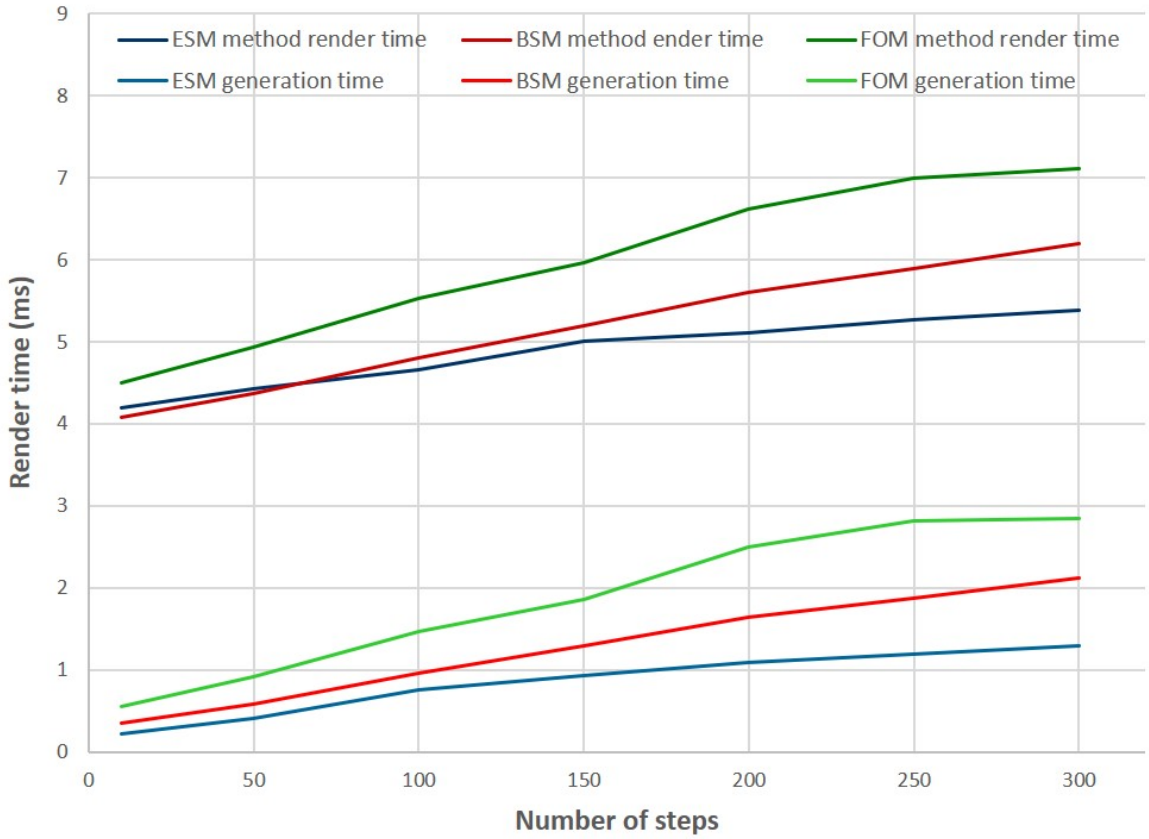


Figure 4.4: Render time of shadow mapping methods with different number of samples.

BSM. If it uses 20 samples, the render time is longest among these methods with some typical settings. Regarding shadow mapping methods, the total render time of ESM is similar to BSM while BSM takes a bit longer time to generate shadow maps. Besides,

FOM is slowest one among the three methods.

Table 4.2: Render time of methods with different number of samples.

Methods	**Total Render Time (ms)**	**Shadow Map Generation Time (ms)**
Secondary Ray Marching (10 steps)	4.9464	-
Secondary Ray Marching (20 steps)	6.0864	-
ESM (50 steps)	4.4330	0.4122
ESM (100 steps)	4.6588	0.7492
BSM (50 steps)	4.3736	0.5836
BSM (100 steps)	4.8072	0.9598
FOM (50 steps)	4.9296	0.9182
FOM (100 steps)	5.5228	1.4678

Some visual results of these methods are displayed in Figure 4.5. In the five images, (a) secondary ray marching is the result of taking 10 steps, (b) ESM and (c) BSM and (d) FOM are using 512×512 shadow maps and takes 100 steps, (e) ground truth is generated by secondary ray marching but takes 1000 steps with a constant step distance (10 metres). The ground truth consumes about 140 milliseconds per frame. When the cloud is distant to another cloud, secondary ray marching cannot cast correct shadow due the the few samples. All the four methods have their features, for example, the shadow area in BSM is closer to the ground truth, but the shadow looks a bit darker. Both ESM and BSM lead to brighter edges of clouds than the ground truth.

4.1.4 Render time with different resolutions

Figure 4.6 displays the results of render time when using different screen resolutions. In all the resolutions, ESM and BSM almost consume the same time, from about 4.5 milliseconds at 720p to about 13.0 milliseconds at 2k. Among the four methods, secondary ray marching takes longest time to render, except at 720p. It takes over 15 milliseconds when render at 2k resolution, which is the longest render time. Regarding FOM, it consumes more time than secondary ray marching at 720p, while it is surpassed by second ray marching at 1080p and 2k.

Figure 4.5: Visual results of tested methods with typical settings.

4.1.5 Render time with different cloud coverage rates

Figure 4.7 presents the render time of the four methods with different cloud coverage rates. The results indicate the render times for all the four methods are in proportion to the cloud coverage rate. While ESM, BSM and FOM shows similar growth rate, secondary ray marching grows faster than shadow mapping methods. When at a low cloud coverage rate, e.g., 10%, secondary ray marching takes shortest time among the four methods, whereas, at a high cloud coverage rate, e.g., 90%, it exceeds all the shadow mapping methods. Secondary ray marching surpasses ESM, BSM and FOM at about 35%, 45% and 78% cloud coverage rates respectively.

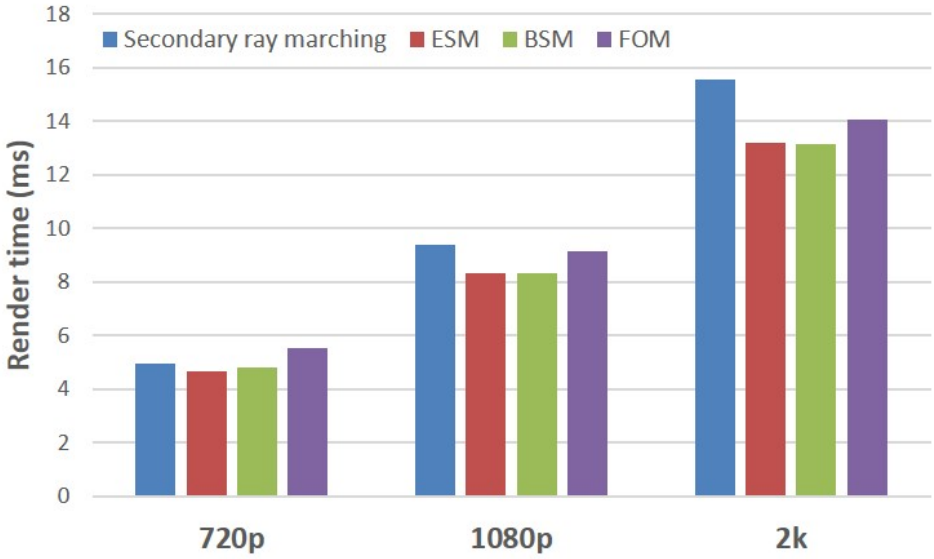


Figure 4.6: Render time with different screen resolutions.

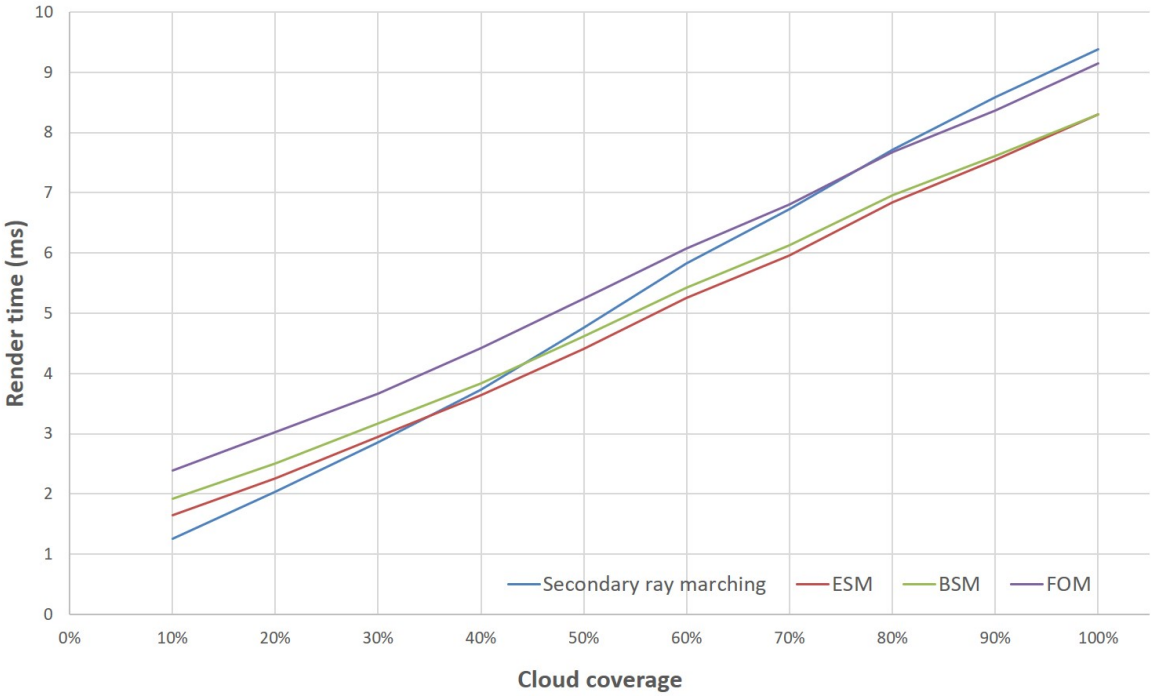


Figure 4.7: Render time with different cloud coverage rates.

4.2 Discussion

Discussions on the performances of the tested methods are described in this section based on the experiment results from the previous section.

Regarding memory consumption, secondary ray marching does not require additional memory, whereas shadow mapping methods do. In every pixel on their shadow maps,

ESM, BSM and FOM store one, three and seven 16 bit float values respectively. The memory footprint among the three methods is FOM consumes more memory than BSM, and both requires more memory than ESM. Hence, when the memory of devices is a bottleneck, we should avoid using those methods consuming lots of memory, such as BSM and FOM. The shadow map resolution also affects memory usage. A 1024×1024 shadow map has four times amount of pixels than a 512×512 shadow map. As a result, higher resolution requires much more larger memory. For instance, 512×512 BSM shadow map takes 12 MB, whereas 1024%1024 BSM shadow map takes 48 MB. Even though higher resolution usually leads to more correct results and less artefacts, the memory consumption may limit using a very large shadow map.

High shadow map resolution also slows rendering. When change shadow map resolution, the difference of the total render time is almost from the shadow map generation time. For example, when using a 512×512 BSM shadow map, its total render time is about 0.5 milliseconds longer than using 256×256 BSM shadow map, while the shadow map generation time is about 0.4 milliseconds longer which is close to 0.5 milliseconds. As for the time to generate shadow maps, there are also some overhead, so when we four times the amount of pixels, the shadow maps generation time is not perfectly four times. For instance, generate FOM shadow map at 512×512 needs approximately 1.4 milliseconds, while the value is approximately 0.7 milliseconds at 256×256. However, with the increment of resolution, the proportion of overhead is getting lower. For all the three shadow mapping methods, the generation time at 512×512 is double to the ones at 256×256, whereas the generation time at 1024×1024 is almost 2.5 times to the ones at 512×512. The effects from computing values stored in the shadow maps are getting obvious. Therefore, to shorten the render time, we should avoid using too high resolutions when generating shadow maps.

Regarding the render time of the methods when using different numbers of steps, more steps usually leads to better visual results, whereas the price may not worth paying. With the increment of steps, the render time increases linearly for all the four methods. Even though the way of sampling is different in secondary ray marching and shadow mapping methods, the render time of secondary ray marching grows fastest. If we use 50 steps, secondary ray marching already consumes about 9.5 milliseconds, which is prohibitive for real-time games. However, if we takes only a few steps, the occlusion of the distant clouds may not be accounted (see Figure 4.5 (a) and (d)). Therefore, takes how many samples when using secondary ray marching is a trade-off between render time and visual results. On the other hand, for the three shadow mapping methods, using over 100 samples may not give us a much better visual results, hence, we can compress the render time by using less steps. However, two less steps (too long step distance) may overlook some

cloud which leads to wrong occlusion, so how to choose the number of steps when using shadow mapping methods should considered when designing game scenes. As for the comparison of the four methods, a summary is presented in Table4.2. When using small amount of steps, secondary ray marching is not very slower than the shadow mapping methods. Therefore, when we do not consider the occlusion of distant clouds, secondary ray marching should be a good choice. Among the shadow mapping methods, ESM and BSM consume similar time while FOM consumes a bit longer time when rendering. According to Figure 4.5, FOM shows more smooth changes on the edge of the clouds than BSM and ESM, while BSM produces closer occlusion results (dark clouds) to the ground truth. Hence, we can choose a method according to our demands on the visual results and styles. For example, when we want to emphasise the shadow cast by clouds, we had better use BSM.

The higher screen resolutions give more burden to secondary ray marching than shadow mapping methods. According to Equation 3.1 3.5, higher screen resolution requires casting more rays. The difference is secondary ray marching has larger N_{light} and the number of pixels will be multiplied by a larger value, while shadow mapping methods have a constant burden N_{SM}. When at 720p, the four methods require similar render time, whereas the secondary ray marching consumes about 2.5 more milliseconds than ESM and BSM at 2K. Since the render time of secondary ray marching will largely surpass the shadow mapping methods at high screen resolutions, we should avoid using secondary ray marching at high screen resolutions. On the other hand, when render at a high screen resolution, e.g. 2k, even the fastest method ESM needs over 12 milliseconds to render a frame. This already not affordable for real-time rendering, so the best way might be render clouds at a low resolution and upsample them to finally display on the higher resolution screen.

Similar to screen resolutions, the coverage of clouds also affects the render time. Secondary ray marching has a larger N_{light} which makes it easier to be affected by $\bar{C}$. In Figure 4.7, the render time of secondary ray marching grows fastest over the cloud coverage rate. Hence, when can decide which methods to use according to the results. For example, secondary ray marching surpasses BSM at about 45% cloud coverage, so when more than 45% of the total pixels are clouds, we should choose BSM than secondary ray marching. However, we also should consider the which visual results we want. Generally, secondary ray marching produces most nature visual results, and when cloud are sparse in the sky, the occlusion of distant clouds could be ignored, so secondary ray marching is usually a good choice when the distribution of cloud is sparse.

Chapter 5

Conclusions and Future Work

The techniques for cloud self-occlusion is investigated in this study. Nowadays, volumetric cloud is already widely used in real-time games, while the evaluation on different techniques for rendering it is less looked into. Cloud self-occlusion is an important part for a realistic appearance of clouds. With correct occlusion, people can better perceive the shape and volume of the clouds. Currently, four methods are usually used for solve the occlusion of cloud: i) secondary ray marching; ii) ESM; iii) BSM; iv) FOM. Even people have proposed the methods, whereas only a few investigated their performance, advantages and disadvantages. Therefore, experiments are designed to estimate the four methods in this study. Secondary ray marching produces more nature appearance and the render time is not very high when only takes a few samples or at a low screen resolution. However, it is not a proper methods to capture the distant occlusion because the limited number of samples. Regarding the rest three shadow mapping methods, they are good at rendering shadow cast by distant cloud, while the attenuation of light travelling in clouds does not look very correct in some cases. The shadow mapping methods have a bit heavier overhead than secondary ray marching because shadow map generation, so they are slower than secondary ray marching in which only a few cloud exists. Whereas, when we need to render to a large screen or render a scene with lots of cloud, shadow mapping methods are faster. In addition, among the three shadow mapping methods, BSM does not consume too much more resource than ESM and usually renders a better result, while FOM takes longest time to render and uses most memory. According to the experiment results, all the methods for cloud occlusion have their own advantages and disadvantages. Hence, engineers can refer this when they are going to implement a sky system in their games.

Even four main cloud self-occlusion techniques have been estimated in this study, there are still a lot more works need to be done for volumetric cloud rendering. The

combinations of different optimisation techniques are not well investigated in this study. To simulate the situations in real-time applications, temporal integration of the scattered light solution are implemented for the experiments. However, the render time under different scales of this optimisation method is not estimated. On top of this, there are other methods to accelerate shadow map generation, such as storing shadow information in look up tables. To better compare the cloud self-occlusion methods, we should measure their performance with different optimisation methods in the future. In addition to occlusion, volumetric cloud has lots of challenges as listed in Chapter 1. For example, when rendering heterogeneous cloud, how different models lead to different appearance is worth estimating. The evaluations on current solutions for these problems are also need to be done in the future.

Bibliography

Annen, T., Mertens, T., Seidel, H.-P., Flerackers, E., and Kautz, J. (2008). Exponential shadow maps. In *Graphics Interface*, pages 155–161. ACM press.

Arbree, A., Walter, B., and Bala, K. (2008). Single-pass scalable subsurface rendering with lightcuts. In *Computer Graphics Forum*, volume 27, pages 507–516. Wiley Online Library.

Bauer, F. (2019). Creating the atmospheric world of red dead redemption 2: a complete and integrated solution.

Beer, A. (1852). Bestimmung der absorption des rothen lichts in farbigen flussigkeiten. *Ann. Physik*, 162:78–88.

Christensen, P. (2008). Point-based approximate color bleeding. *Pixar Technical Notes*, 2(5):6.

Clausse, R. and Facy, L. (1961). *The Clouds*. Evergreen Books, 1th edition.

Decreusefond, L. and Üstünel, A. S. (1998). Fractional brownian motion: theory and applications. In *ESAIM: proceedings*, volume 5, pages 75–86. Citeseer.

Deng, H., Wang, B., Wang, R., and Holzschuch, N. (2020). A practical path guiding method for participating media. *Computational Visual Media*, 6(1):37–51.

Guerrette, K. (2014). Moving the heavens: An artistic and technical look at the skies of the last of us. *Game Developers Conference*.

Harris, M. J. and Lastra, A. (2002). Real-time cloud rendering for games. In *Proceedings of Game Developers Conference*, pages 21–29.

Henyey, L. G. and Greenstein, J. L. (1941). Diffuse radiation in the galaxy. *The Astrophysical Journal*, 93:70–83.

Herholz, S., Zhao, Y., Elek, O., Nowrouzezahrai, D., Lensch, H. P., and Křivánek, J. (2019). Volume path guiding based on zero-variance random walk theory. *ACM Transactions on Graphics (TOG)*, 38(3):1–19.

Hillaire, S. (2015). Towards unified and physically-based volumetric lighting in frostbite. *Proc. SIGGRAPH Advances Real-Time Rendering Course*.

Hillaire, S. (2016). Physically based sky, atmosphere and cloud rendering in frostbite. *Physically Based Shading in Theory and Practice, SIGGRAPH*.

Hillaire, S. (2020). Physically based and scalable atmospheres in unreal engine. *Physically Based Shading in Theory and Practice, SIGGRAPH*.

Hufnagel, R. and Held, M. (2012). A survey of cloud lighting and rendering techniques.

Jakob, W. and Marschner, S. (2012). Manifold exploration: A markov chain monte carlo technique for rendering scenes with difficult specular transport. *ACM Transactions on Graphics (TOG)*, 31(4):1–13.

Jansen, J. and Bavoil, L. (2010). Fourier opacity mapping. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 165–172.

Jarosz, W., Nowrouzezahrai, D., Sadeghi, I., and Jensen, H. W. (2011). A comprehensive theory of volumetric radiance estimation using photon points and beams. *ACM transactions on graphics (TOG)*, 30(1):1–19.

Jensen, H. W. and Christensen, P. H. (1998). Efficient simulation of light transport in scenes with participating media using photon maps. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 311–320.

Kajiya, J. T. and Von Herzen, B. P. (1984). Ray tracing volume densities. *ACM SIGGRAPH computer graphics*, 18(3):165–174.

Kallweit, S., Müller, T., Mcwilliams, B., Gross, M., and Novák, J. (2017). Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks. *ACM Transactions on Graphics (TOG)*, 36(6):1–11.

Kaplanyan, A. (2009). Light propagation volumes in cryengine 3. *ACM SIGGRAPH Courses*, 7(2).

Keller, A. (1997). Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56.

Křivánek, J. and d'Eon, E. (2014). A zero-variance-based sampling scheme for monte carlo subsurface scattering. In *ACM SIGGRAPH 2014 Talks*, pages 1–1.

Křivánek, J., Georgiev, I., Hachisuka, T., Vévoda, P., Šik, M., Nowrouzezahrai, D., and Jarosz, W. (2014). Unifying points, beams, and paths in volumetric light transport simulation. *ACM Transactions on Graphics (TOG)*, 33(4):1–13.

Lafortune, E. P. and Willems, Y. D. (1996). Rendering participating media with bidirectional path tracing. In *Eurographics Workshop on Rendering Techniques*, pages 91–100. Springer.

Lee, A. (2012). Life of pi. `https://www.20thcenturystudios.com/movies/life-of-pi`. Accessed: 2022-07-24.

Liang, Y., Wang, B., Wang, L., and Holzschuch, N. (2019). Fast computation of single scattering in participating media with refractive boundaries using frequency analysis. *IEEE Transactions on Visualization and Computer Graphics*, 26(10):2961–2969.

Novák, J., Nowrouzezahrai, D., Dachsbacher, C., and Jarosz, W. (2012a). Progressive virtual beam lights. In *Computer Graphics Forum*, volume 31, pages 1407–1413. Wiley Online Library.

Novák, J., Nowrouzezahrai, D., Dachsbacher, C., and Jarosz, W. (2012b). Virtual ray lights for rendering scenes with participating media. *ACM Transactions on Graphics (TOG)*, 31(4):1–11.

Perlin, K. (2002). Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682.

Pharr, M., Jakob, W., and Humphreys, G. (2016). *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

Preetham, A. J., Shirley, P., and Smits, B. (1999). A practical analytic model for daylight. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100.

Rockstar (2018). Red dead redemption 2. `https://www.rockstargames.com/reddeadredemption2/`. Accessed: 2022-07-24.

Schneider, A. (2017). Nubis: authoring real-time volumetric cloudscapes with the decima engine. *SIGGRAPH Advances in Real-Time Rendering in Games Course, ACM*, pages 619–620.

Schneider, A. and Vos, N. (2015). The real-time volumetric cloudscapes of horizon: Zero dawn. *Advances in Real-time Rendering, SIGGRAPH*.

Wang, B., Gascuel, J.-D., and Nolzschuch, N. (2016). Point-based light transport for participating media with refractive boundaries. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*, pages 109–119.

Weber, P., Hanika, J., and Dachsbacher, C. (2017). Multiple vertex next event estimation for lighting in dense, forward-scattering media. In *Computer Graphics Forum*, volume 36, pages 21–30. Wiley Online Library.

Worley, S. (1996). A cellular texture basis function. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 291–294.

Wrenninge, M., Kulla, C. D., and Lundqvist, V. (2013). Oz: the great and volumetric. In *SIGGRAPH Talks*, pages 46–1.

Wronski, B. (2014). Volumetric fog: Unified compute shader based solution to atmospheric scattering. *Advances in Real-time Rendering, SIGGRAPH*.

Wu, W., Wang, B., and Yan, L.-Q. (2022). A survey on rendering homogeneous participating media. *Computational Visual Media*, 8(2):177–198.

Yusov, E. (2014). High-performance rendering of realistic cumulus clouds using precomputed lighting. In *High Performance Graphics*, pages 127–136.