Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Atmosphere and Cloud Rendering in Real-time

Matěj Sakmary

Supervisor: Ing. Jaroslav Sloup
May 2022

Acknowledgements

I would like to thank everyone who has supported me during the creation of this work. I would also like to thank Ing. Jaroslav Sloup, my supervisor, for offering a lot of valuable advice and guidance throughout the whole process.

Declaration

I declare that I have created the presented thesis independently and that I have quoted all used sources of information in accordance with Methodical instructions about ethical principles for writing academic theses.

Abstract

This work presents an implementation of the rendering of the volumetric cloud and atmosphere. The implementation aims to combine and slightly alter previously published solutions to produce a single unified look. Raymarching was used as the main method to render both the atmosphere and clouds. An approach using multiple precomputed Look-up tables (LUTs) proposed by Sébastien Hillaire was used to render the atmosphere. This was combined with the option to render volumetric clouds using pre-computed three dimensional texture setup storing procedurally generated noise described by Andrew Schneider. The final solution is able to render images in high dynamic range before applying post-processing effects and using adaptive luminance to transform the image into LDR for presentation.

Keywords: Volumetric clouds, Realtime rendering, Atmosphere, Vulkan

Supervisor: Ing. Jaroslav Sloup
Praha 2,
Karlovo náměstí 13,
E-413

Abstrakt

Tato práce představuje implementaci renderování volumetrických mraků a atmosféry. Cílem implementace bylo zkombinovat a mírně pozměnit dříve publikovaná řešení s cílem vytvořit jednotný vzhled. Metoda zvaná Raymarching byla použita pro renderování atmosféry a mraků. Přístup, využívající množství předpočítaných Look up tabulek navržen Sébastienem Hillairem, byl použit k renderování atmosféry. Toto bylo zkombinováno s možností vykreslování volumetrických mraků za použití trojrozměrných textur, ve kterých je uložen procedurálně generovaný šum, což bylo popsáno Andrew Shneiderem. Konečné řešení je schopno vykreslovat snímky ve vysokém dynamickém rozsahu, na které jsou aplikovány post procesové efekty. Tyto snímky jsou poté pomocí adaptivní luminance transformovány do nízkého dynamického rozsahu pro prezentaci na obrazovku.

Klíčová slova: Volumetrické mraky, Realtime renderování, Atmosféra, Vulkan

Překlad názvu: Vykreslování oblohy a mraků v reálném čase

Contents

Figures

Tables

Chapter 1

Introduction

Figure 1.1: Image rendered using our application

Having a realistic and believable atmospheric model when rendering dynamic environments in games or other interactive applications is an important part of creating virtual worlds. The sky, atmosphere, and cloud configuration can instantly change the mood of the scene. This is especially important for applications that require dynamic time of day and weather. Additionally, these effects are also interconnected and affect each other, making them even harder to simulate. Despite the gradual increase of compute power available in personal computers and laptops simulating the complex light interactions producing visual appearance of sky and clouds is still very difficult. Paired with the constraint of displaying such effects in real time makes this problem even harder.To avoid these problems, we are forced to adapt number of approximations, sacrificing a small amount of visual quality but in turn gaining a big reduction in the problem complexity.

1.1 Goals

The goal of this work was to provide a complete solution to show dynamic clouds and the sky in real time. To achieve this, we try to combine multiple well-described techniques into a single solution. Additionally, we wanted to have the option to modify most of the sky and cloud look directly inside of the application without big delays or freezes.

Due to the already large scope of our work, we decided that the whole

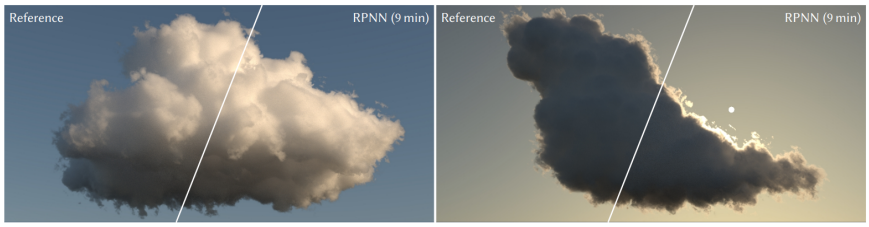


Figure 1.2: Image taken from [KMM+17] showing the cloud rendered using their proposed method combining Monte Carlo integration with neural network.

frame-time budget will be dedicated to the rendering of desired effects. Our initial aim was to render previously described effects in 60 frames per second throughout most of the application execution. Later we decided to lower this value to 30 frames per second in order to produce better looking results. It is important to note that the application frame rate is highly dependent on the parameterization and the desired quality of the final image. This means that even though the application frame rate can fall below 60 fps, it is still possible to trade some of the visual fidelity for the sake of increased performance.

1.2 Previous work

Now we will provide an overview of previous research that involved rendering of the atmosphere, sky, and clouds. We will describe multiple possible approaches and reason about their advantages and disadvantages.

The most physically accurate method to render volumetric effects such as sky and clouds is to use path tracing. This method sends rays from the camera and follows them as they bounce when hitting objects in the scene until they reach a light source. Although using this method produces the best looking effects, the computational complexity is very high. This is mainly given by the amount of rays we need to send from the camera to reduce the noise in the final image.

Recently, there have been several promising advances in this field, mainly combining path-tracing with neural networks. These neural networks serve mainly to denoise the resulting image, reducing the amount of needed rays. A method was also proposed that uses a neural network specifically to render volumetric clouds [KMM+17]. The neural network pre-learns spatial flux distribution and later predicts the radiance function for each shading configuration.

Although the proposed improvement reduces the computation time, it is still very difficult to achieve real-time frame rates while maintaining the desired image quality using these methods. This is why they are mainly used in offline rendering or as a ground truth when developing new methods.

Another approach using fitted mathematical models has also been proposed ([HW12] [WVBR+21]). Methods leveraging this principle usually use measured data in order to build a set of parameters used to evaluate the sky look. These models tend to be very fast; however, due to the dependence on the

data measured in the real world, these methods fail to provide the option to change the parameters of the atmosphere. Additionally, when the parameters of the atmosphere are changed, a new model has to be fitted, which is not possible to do in real time. As the goal of this work is rendering parameterizable atmospheres reflecting the conditions present on other planets as well as the ones on Earth, these methods did not fit our requirements.

Finally, many methods use ray marching to achieve their results. Offering a good compromise between physical accuracy and speed, it is very popular in problems requiring rendering volumetric effects such as clouds, mist, or atmospheres. Unlike path-tracing, ray marching does not spawn additional rays. Instead, multiple steps are taken along a ray, sampling the medium at each step. These medium samples are then used to calculate the final look of the ray-marched medium. Because of those reasons, we chose to use ray marching in our implementation as well.

1.2.1 Sky and atmosphere

Multiple methods have previously been proposed for rendering physically based atmospheres in real time. The first ones focused only on single scattering evaluation by ray marching the atmosphere from viewpoint for each pixel on the screen. Although evaluating only singly scattered light has performance benefits, it struggles to represent realistic looking atmospheres. This is most visible when trying to represent more dense atmospheres, where evaluating single scattering only results in dark and unrealistic scenes. Due to this, methods taking into account multiple scattering were introduced ([BN08], [Yus14a]).

Such methods need to rely on precomputing parts of the rendering equation and storing them into 2D, 3D, and 4D tables called Lookup Tables(LUTs) in order to speed up certain parts of the evaluation. This improves the rendering time significantly because where previously the same evaluation was repeated tens or hundreds of times, it is now only computed once at the beginning and the results are then accessed whenever needed, which is much faster. The main drawback of these methods is the inability to change the atmosphere parameters in real time. Whenever the atmosphere parameters change, all of the LUTs used have to be recalculated, which is a very expensive operation. This results in either a long delay before seeing the changes caused by time-slicing the operation or worse in the whole application freezing until the LUT computations are finished. The second disadvantage is that, because the results are obtained by ray marching each pixel, the performance is tied to the resolution of the screen.

When trying to solve these problems [Hil20] introduced a new method. The first proposal was a new method to evaluate multiple scattering inspired by a dual scattering approximation used when simulating multiple scattering effects in hair.This reduced the time to precompute LUTs greatly making it possible to update atmosphere parameters with almost no delay. The second proposal was to precompute the final sky view and the aerial perspective into fixed-size latitude/longitude textures, which are later sampled and upscaled.

Figure 1.3: Result cloud obtained using method described by [BNM+08]

This effectively decouples the computation complexity from window resolution and introduces additional speed improvements.

1.2.2 Clouds

Many approaches to rendering clouds have been described. Now we will try to summarize the ones that we found most interesting or relevant to our work. One possible approach to rendering volumetric clouds was to represent them as volumes of particles. For example, a particle-based rendering method was previously presented by Yusov [Yus14b]. The clouds were modeled using copies of randomly rotated and scaled copies of a single reference particle. The complex optical properties of this reference particle were precomputed during preprocessing, making this process possible to use in real-time applications.

Figure 1.4: Rendered clouds obtained by Yusov et al. [Yus14b] using the reference particle based technique.

Another possible technique was presented by Bouthors et al. [BNM+08]. By combining meshes to represent low resolution cloud boundaries together with procedural volumetric hyper-textures which add the detail under the mesh boundary an efficient cloud representation was achievable. When rendering, the cloud surface is covered with circular collectors, which are used to evaluate the incoming light. Using this information along with a set of pre-computed

trasfer tables the light is integrated. This produces very good-looking clouds; however, the cloud representation is not trivial to tweak. This, along with the pretty high overall complexity of the described method, was the main reason why different, simpler methods were developed.

The most recent work by Schneider [SV15] uses a fully procedural set of volumetric noise textures to produce similar results. These noise textures are used to represent changes in density in a medium caused by clouds. The clouds are then rendered by ray marching the cloud volume and sampling the medium. This method also allows us to completely change the overall look of the cloud layer by only tweaking a few parameters. Additionally, it is possible to simulate dynamic lighting conditions caused, for example, by changing the time of day. This was a good fit for our purpose and goals, so we decided to use this method in our implementation.

Chapter 2

Physical model

This section was heavily inspired and contains parts of [Pet06],[Hil20], and [CBE+21] that were distilled to contain only parts relevant to this work.

When electromagnetic radiation (photons of light) travel through the atmosphere, they collide with molecules the atmosphere consists of. During this collision, part of the energy carried by the radiation is absorbed, part is reflected (scattered), and part is emitted.

2.1 Extinction, Scattering and Absorption Coefficients

The amount of extinct, scattered and absorbed energy is given by the respective *extinction, scattering and absorption coefficients* denoted as β_e ,β_a and β_s. *Absorption coefficient* is defined as

$$\beta_a = \frac{4\pi n_i}{\gamma} \tag{2.1}$$

where γ is the wavelength of the radiation in vacuum and n_i is the complex part of *index of refraction*. Thus, this coefficient denotes the rate of energy attenuation per unit of distance at a point x. Similarly, we define a scattering coefficient. Extinction coefficient is then defined by the sum of the absorption and scattering coefficients.

$$\beta_e = \beta_a + \beta_s \tag{2.2}$$

2.2 Extinction over a finite path

Whenever radiation passes through a medium, its intensity is attenuated exponentially by the extinction coefficient represented by the following relationship.

$$L(\gamma, x) = I(\gamma, 0) \exp(-\beta_e x) \tag{2.3}$$

where γ is the wavelength of the energy considered. Equation (3) holds under the assumption that the extinction coefficient is uniform through the

atmosphere, which is rarely true. To correctly compute attenuation over a path where the extinction coefficient varies, integrating the coefficient along the entire path of the ray is more suitable. So, to obtain the amount of light that arrives at the point x_2 from the point x_1 given the intensity of light at this starting point and *extinction coefficient* β_e is given by the following relationship.

$$L(\gamma, x_2) = L(\gamma, x_1) \exp\left[- \int_{x_1}^{x_2} \beta_e(x)\, dx \right] \tag{2.4}$$

Here, the integral quantity inside the brackets is called the optical path τ between the points x_1 and x_2

$$\tau(x_1, x_2) = \int_{x_1}^{x_2} \beta_e(x)\, dx \tag{2.5}$$

By exponentiation of the optical path τ, we get *transmittance* denoted by T between x_1 and x_2

$$T(x_1, x_2) = e^{-\tau(x_1, x_2)} \tag{2.6}$$

We can then rewrite the previous equation of light from point x_1 arriving at point x_2 as

$$L(\gamma, x_2) = T(x_1, x_2) I(\gamma, x_1) \tag{2.7}$$

Rewriting equations (4) and (7) in a more commonly used form in computer graphics gives us

$$L(\gamma, x, \overrightarrow{\omega}) = T(x, x_0) L(\gamma, x_0, -\overrightarrow{\omega}) \tag{2.8}$$

Where $\overrightarrow{\omega}$ is the direction of the ray being evaluated, x is the starting point of this ray, and x_0 is the respective end point of this ray.

2.3 Scattering effects of the atmosphere

So far, the scattering effects of atmosphere particles have only been considered by direct attenuation of energy over a path radiation takes through the atmosphere. However,unlike absorption effects, when energy is scattered away, it is not converted to another form such as heat or chemical energy, but instead added back to the atmosphere. This means that the loss of energy along one considered path is added along other paths through the atmosphere.

However, the energy scattered away is not uniform in all possible directions. To represent this, we use a *scattering phase function* denoted $P(\overrightarrow{\omega}', \overrightarrow{\omega})$ where $\overrightarrow{\omega}'$ is the incoming direction and $\overrightarrow{\omega}$ is the direction of interest (the direction of the ray along which we calculate).

Taking this into account, the scattering formula is denoted as follows.

$$dL_{scat}(\gamma, x, \overrightarrow{\omega}) = \frac{\beta_s}{4\pi} \int_{4\pi} P(\overrightarrow{\omega}', \overrightarrow{\omega}) L(\gamma, x, \overrightarrow{\omega}')\, d\overrightarrow{\omega}'\, ds \tag{2.9}$$

Here, the integral is over 4π steradians of the solid angle, and the result is weighed by the scattering coefficient. To be physically accurate, the concept

of energy conservation needs to be satisfied (the amount of out-scattered energy cannot be greater than the amount of incoming energy). This is why the equation is normalized by 4π. In the later section this normalization will be emitted, thus assuming normalized scattering phase function, meaning the following must be true.

$$\int_{4\pi} P(\overrightarrow{\omega}', \overrightarrow{\omega})\, d\overrightarrow{\omega}' = 1 \tag{2.10}$$

When considering an atmosphere, a simplification can be made. Because particles in the atmosphere are spherical or randomly oriented, the scattering phase function can be rewritten to depend only on one parameter Θ between the original direction and the scattered direction. The relationship between those two parameterizations is given as follows.

$$\cos\Theta = \overrightarrow{\omega}' \cdot \overrightarrow{\omega} \tag{2.11}$$

which simplifies (9) to

$$dL_{scat}(\gamma, x, \overrightarrow{\omega}) = \int_{4\pi} \beta_s(y) P(cos\Theta) L(\gamma, x, \overrightarrow{\omega}')\, d\overrightarrow{\omega}'\, ds \tag{2.12}$$

Note that β_s needs to be placed inside of the first integral and is made a function of y since it depends on the distribution of particles inside the atmosphere, which can change with position. This will be explained in more detail later.

Applying concepts described in the previous section, the equation for calculating the scattering contribution to a path along a ray with direction $\overrightarrow{\omega}$ arriving at a point x can be computed as follows:

$$L_{scat}(\gamma, x, \overrightarrow{\omega}) = \int_{x}^{x_0} \beta_s(y) T(x, y) \int_{4\pi} P(cos\Theta) L(\gamma, y, \overrightarrow{\omega}')\, d\overrightarrow{\omega}'\, dy \tag{2.13}$$

2.4 Scattering and extinction effects of clouds

Clouds consist of a high concentration of very small water droplets, which generally have spherical shape. Because the average radius of water droplets in a cloud is usually around 10 μm, which is much higher than the wavelength of the incoming visible light (380 - 700 nm) we only consider Mie scattering to occur inside of a cloud (described later in Section 2.7.2).

Most of the concepts described above also apply to clouds. The main difference being the much higher scattering and absorption coefficient given by the higher density of particles inside clouds. We use the scattering and absorption coefficients measured by [HKS98] together with the computations provided by [Kok04]. Please note that these coefficients can also be calculated with the help of equations described in Section 2.7.2.

This means that to evaluate the direct contribution of sunlight to the overall luminance can be calculated using Equation 2.8. Most of the final

white look of clouds is caused by the complex scattering effects inside of a cloud. This is due to the high scattering coefficient which is why, in contrast to atmospheric computation, multiple scattering effects of clouds are much more important. Similarly to the direct light contribution, the scattering contribution can be calculated using the previously described equation 2.13.

2.5 The rendering equation

If we sum the effects described above by (7) and (12), we get the following form.

$$L(\gamma, x, \overrightarrow{\omega}) = \underbrace{T(x, x_0)L(\gamma, x_0, -\overrightarrow{\omega})}_{direct\ light\ from\ sun} + \underbrace{\int_x^{x_0} \beta_s(y)T(x,y) \int_{4\pi} P(cos\Theta)L(\gamma, y, \overrightarrow{\omega}')\, d\overrightarrow{\omega}'\, dy}_{in-scattered\ light\ along\ the\ ray} \tag{2.14}$$

Because we want to be able to also render effects of opaque objects and thus render volumetric shadows cast by those objects onto the atmosphere, we also need to add a *visibility term* denoted by V introduced by [Hil20]. This also enables us to render the effect of a planet as a whole casting a shadow onto the atmosphere, which can be visible during dusk and sunset as a darker band in the atmosphere just above the planet's surface. This Visibility coefficient is added to the single scattering evaluation and is either 0, when the light source is not visible, or 1 when it is visible.

When the currently evaluated ray hits the ground, the effect of radiant energy reflected from this point along the ray must also be taken into account. As previously described by [BN08] and extended by [CBE+21], this energy is calculated by integrating all the energy arriving at the intersection point s_0 over the hemisphere (2π steradians of solid angle) that is positioned above the tangent plane of such a point. This energy is then weighted by the *bidirectional reflectance distribution function (BRDF)* denoted as $f(s_0, \hat{\Omega}', \hat{\Omega})$ of the planet's surface and the cosine between the normal at the point s_0 and the ray incoming direction $\hat{\Omega}'$ [CBE+21] giving us following formula

$$R(\gamma, x, \overrightarrow{\omega}) = T(x, x_0) \int_{2\pi} f(x_0, \overrightarrow{\omega}', \overrightarrow{\omega})(n(x_0) \cdot \overrightarrow{\omega}')L(\gamma, x_0, \overrightarrow{\omega}') \tag{2.15}$$

Finally, adding all of this together results in a full rendering equation formula.

$$
\begin{aligned}
L_{tot}(\gamma, x, \overrightarrow{\omega}) = & \underbrace{T(x, x_0) L(\gamma, x_0, -\overrightarrow{\omega})}_{direct\ light\ from\ sun} \\
& + \underbrace{T(x, x_0) \int_{2\pi} f(x_0, \overrightarrow{\omega}', \overrightarrow{\omega})(n(x_0) \cdot \overrightarrow{\omega}') L(\gamma, x_0, \overrightarrow{\omega}')}_{light\ reflected\ from\ surface} \\
& + \underbrace{\int_x^{x_0} \beta_s(y) T(x, y) \int_{4\pi} P(cos\Theta) L(\gamma, y, \overrightarrow{\omega}')\, d\overrightarrow{\omega}'\, dy}_{in-scattered\ light\ along\ the\ ray}
\end{aligned}
\tag{2.16}
$$

2.6 Multiple scattering approximation

As described above, calculating only single scattering of light is not sufficient for realistic results. This is because the scattering of light inside an atmosphere occurs multiple times, so only considering a single scatter results in a big loss of calculated energy. When calculating multiple scattering model proposed by [CBE+21] and [BN08] uses an iterative approach which can be mathematically denoted as follows:

$$
\begin{aligned}
L &= L_0 + (R + S)[L_0] + (R + S)[(R + S)[L_0]] + \dots \\
&= L_0 + L_1 + L_2 + \dots = L_0 + L_*
\end{aligned}
\tag{2.17}
$$

where L_0 is the attenuated direct sunlight that reaches the considered point. The big disadvantage of this model is the high computational cost. This quickly becomes a problem for denser atmospheres, where multiple scattering of order up to 40 and more has to be evaluated. The solution to this proposed in [Hil20] is also used in this work.

The overall smooth distribution of participating media in the atmosphere allows the assumption that the energy reaching a single point in space is the same for all points within a large area around it for scattering orders greater than 2. Because of this, the multiple scattering evaluation can be simplified.

First, second-order scattering is evaluated according to the methods described previously.

$$
L_{2^{nd}order} = \int_{4\pi} L_{tot}(\gamma, x, -\overrightarrow{\omega}) P(cos\Theta) \tag{2.18}
$$

However, instead of integrating luminance, a unitless placeholder factor is used L_I. The reason why this is done will be explained later.

Secondly also untiless factor of f_{ms} representing the transfer of energy that would occur from the area around and aimed towards considered point is calculated by

$$
f_{ms} = \int_{4\pi} L_f(\gamma, x, -\overrightarrow{\omega}) P(cos\Theta)\, d\overrightarrow{\omega} \tag{2.19}
$$

$$
L_f(\gamma, x, -\overrightarrow{\omega}) = \int_x^{x_0} \beta_s(y) T(x, y) 1\, dy \tag{2.20}
$$

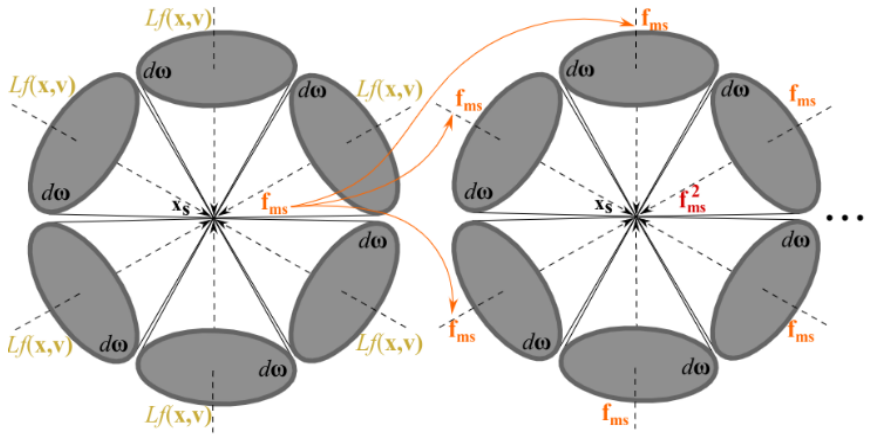


Figure 2.1: Image provided by [Hil20] showing how the F_{ms} is computed by usingf_{ms}.

Because f_{ms} is in the range between 0 and 1 and given the previous assumption about the amount of energy reaching a point for scattering orders greater than two, the infinite multiple scattering energy contribution can be approximated by

$$F_{ms} = 1 + f_{ms} + f_{ms}^2 + f_{ms}^3 + \dots \tag{2.21}$$

Given that the value of f lies in the interval [0,1], equation (21) can be solved analytically.

$$F_{ms} = \frac{1}{1 - f_{ms}} \tag{2.22}$$

Because the placeholder unitless factor was used when calculating the second order of the in-scattered energy, the total contribution of a directional light with an infinite number of scattering orders can be evaluated as

$$\Psi_{ms} = L_{2^{nd}order} F_{ms} \tag{2.23}$$

Finally, Equation (13) should now be modified to give the final form.

$$L_{scat}(\gamma, x, \overrightarrow{\omega}) = \int_x^{x_0} (\beta_s(y) T(x,y) P(cos\Theta) + \Psi_{ms}) L(\gamma, y, \overrightarrow{\omega}')\, dy \tag{2.24}$$

where $\overrightarrow{\omega}'$ is the direction from the current position y to the considered light source.

2.7 Physical properties of particles inside atmosphere

As already mentioned, incoming light in form of electromagnetic radiation collides with particles that make up the atmosphere. The properties of those collisions depend on the wavelength of the incoming radiation, but they also depend on the size of the particle considered. When considering the visible

wavelength band (440nm - 680nm), which is the only band in which this work is interested, the scattering effects can be described by two underlying theories, Mie and Rayleigh Scattering.

2.7.1 Scattering and absorption effects of molecules and atoms

When considering small particles (that is, particles much smaller than the wavelength of the incident radiation $r_{particle} \ll \gamma_{incident\ radiation}$) Rayleigh scattering theory is used. Rayleigh scattering phase function is used to approximate the angular distribution of the outscattered radiation. The more complicated model proposed by [CBE+21], which takes into account the anisotropy of the atmosphere molecules, is used.

$$P_R(\theta) = 0.7629(1 + 0.932 \cdot cos^2(\theta)) \cdot \frac{1}{4\pi} \tag{2.25}$$

Note that normalization by 1 over 4 pi is included, as mentioned above. For the scattering coefficient, once again the one proposed by [CBE+21] is used.

$$\beta_R^{scat}(h, \gamma) = \frac{8\pi^3(n(\gamma)^2 - 1)^2}{3N\gamma^4} \cdot e^{-\frac{h}{H_R}}\ [m^{-1}] \tag{2.26}$$

Here, $n(\gamma)$ and N are the refractive index of the medium (whose imaginary part was also used in Eq. (1) when describing the absorption index β_a) and the molecular density at sea level, respectively. Also, note the term $e^{-\frac{h}{H_R}}$ that describes the variation of atmospheric density according to height h and scale height H_R. The change in the density of the atmosphere and thus the change in the scattering coefficient is why in Equation (12) the term β_s was put inside the integral and made dependent on the position in the atmosphere.

One more thing to notice is the dependency of the Rayleigh phase function on the wavelength $1/\gamma^4$, which means that shorter wavelengths are scattered more than long wavelengths. This is the reason why the sky is blue during the day because the path the incoming light takes is quite short, and so blue-light scattering is more prevalent. On the other hand, during dusk or sunset the path of the light through the atmosphere, especially near the horizon, is long, scattering almost all of the blue light away so that only reddish wavelengths remain.

Rayleigh theory does not consider the absorption of small particles. However, as was shown above (Section 2.1), when the complex refraction index is known, the absorption effect can still be calculated. The model used in this work (once again proposed by [CBE+21]) is slightly more complicated than the one described above. It specifies a generic description for the absorption coefficient.

$$\beta_R^{abs}(\gamma, r) = \frac{8\pi^2 r^3 N}{\gamma} \cdot Im\left(\frac{n(\gamma)^2 - 1}{n(\gamma)^2 + 2}\right) \cdot e^{-\frac{h}{H_R^{particle}}}[m^{-1}] \tag{2.27}$$

For more details on how to calculate the absorption coefficient when the complex refractive index N is not known, please refer to [CBE+21]. Previously described calculations can be used to add additional molecules and particles not present in the currently described models.

It is especially important to simulate the absorption effects of molecules such as Ozone or Oxygen. Because ozone has a high absorption coefficient, especially at wavelengths around green, it is responsible for the deep blue color of the sky during sunset or dusk. This effect can be seen in 2.2.

2.7.2 Scattering and absorption effects large particles

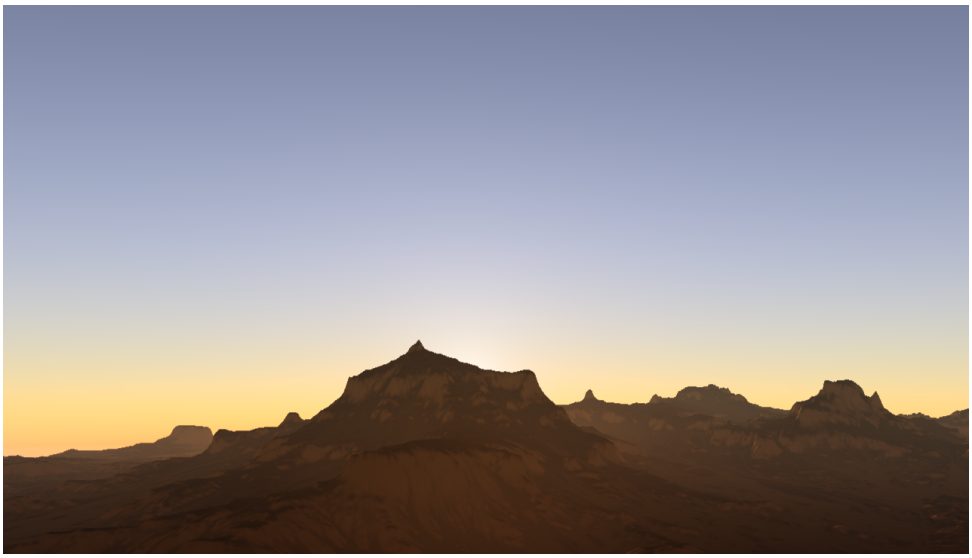
When considering particles that are comparable or larger than the wavelength of the incident radiation, such as dust or water droplets, Mie's theory must be used to produce realistic looking results. Unlike the smaller particles considered above, larger particles tend to scatter strongly forward. This results in the visible haze around the Sun when the atmosphere is thickened by, for example, rain or cloud cover (because clouds are just droplets of water bunched together). The double Henyey-Greenstein phase function approximation proposed by [CBE+21] is used in this model.

$$P_M(\theta, g_1(\gamma), g_2(\gamma), \alpha(\gamma)) = \left[\alpha \frac{(1+g_1^2(\gamma))}{(1+g_1^2(\gamma)-2g_1(\gamma)cos(\theta))^{\frac{3}{2}}} + (1-\alpha)\frac{(1+g_1^2(\gamma))}{(1+g_2^2(\gamma)-2g_2(\gamma)cos(\theta))^{\frac{3}{2}}}\right] \cdot \frac{1}{4\pi} \tag{2.28}$$

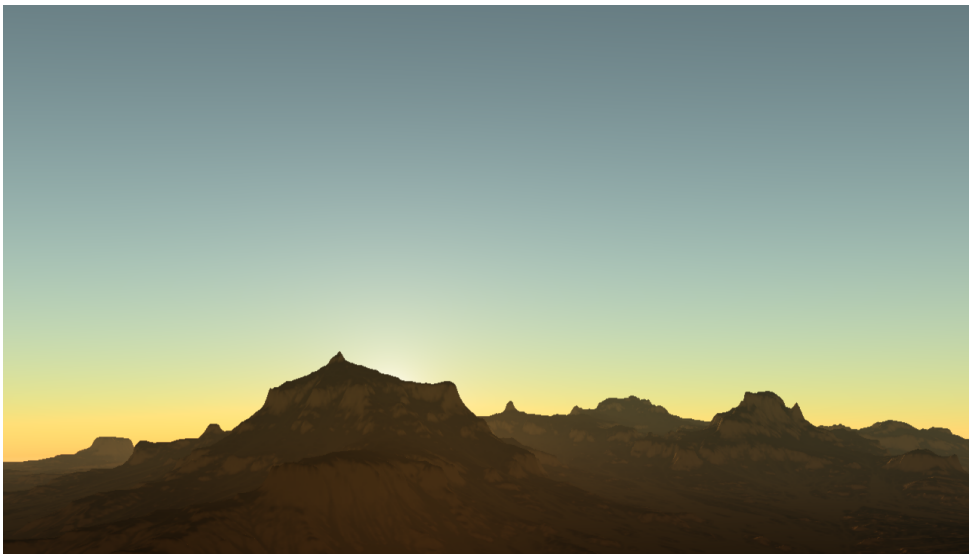
Mie scattering coefficient is approximated by

$$\beta_M^{scat}(h,\gamma) = 0.434C(T)\pi\left(\frac{2\pi}{\gamma}\right)^{v-2} K \cdot e^{-\frac{h}{H_M}} \tag{2.29}$$

For more details on scattering or extinction coefficients, see [CBE+21].



(a) : Sunset with ozone absorption.



(b) : Sunset without ozone absorption greenish tint can be visible.

Figure 2.2: Images showing the difference the absorption effect of ozone and oxygen makes on the atmosphere look during sunset.

Chapter 3

Proposed solution

First we describe the high-level overview of everything that happens during one frame. Along with this, we also provide additional details regarding the Look up table setup and parameterization. After that, multiple chapters will follow, describing every step from the start all the way to the final image in more detail.

The process of rendering a single frame can be divided into four parts. These four parts directly map to four command buffers which are submitted to the GPU each frame. The first command buffer is responsible for the computation of the four LUTs used to render the atmosphere. The second command buffer renders all the objects and terrain. Along with this the atmosphere, it's effects and clouds are also rendered. After this, the third command buffer is executed. The main function of this command buffer is to map values from the HDR range into LDR that is used by the image presented to the screen. Lastly, the fourth command buffer containing commands used to render the UI controlling various parameters of the atmosphere and clouds.

We have one more command buffer responsible for the computation of the Worley noise texture described in Section 3.2. Since this computation can be expensive, we decided not to submit this buffer each frame but instead only during the first frame.

3.1 Atmosphere Precomputations

As mentioned above, to speed up the time taken to render the atmosphere, it is beneficial to precompute certain parts of the rendering equation and store them in multidimensional tables. In this work, an LUT setup proposed by [Hil20] is used. The following section will provide a description of this model.

3.1.1 Transmittance LUT

First introduced by [BN08] this LUT is used to store the Transmittance T described in Section 2.2 (specifically Equation 2.6). When the atmosphere is ray-marched, the value of T is used very frequently to model the atmosphere attenuating the passing light. However, to compute this value, a second ray must be traced towards the light source, which can be very expensive. Given

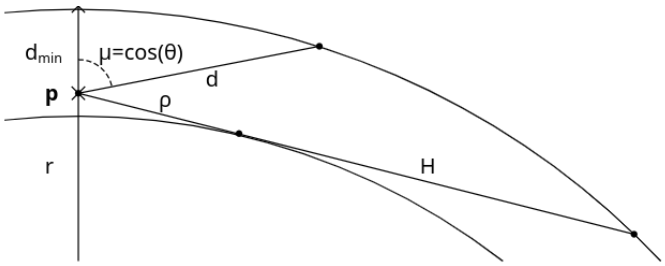


Figure 3.1: Image taken from [BN08] showing the mapping of Transmittance LUT

the above conditions and the overall smooth distribution of atmospheric transmittance it is beneficial to precompute Transmittance value for the entire atmosphere and store it into a table.

In this model, the spherical symmetry of the atmosphere is considered. This means that the density of the atmosphere (and thus the scattering and absorption parameters of the atmosphere) only changes with height. Given this assumption, we store the precomputed values into a 2D lookup table. This LUT is parameterized as follows:

- x axis - map x to a value between [0,1] by considering the distance d to the top atmosphere boundary, compared to its minimum and maximum values $d_{\text{min}} = r_{\text{top}} - r$ and $d_{\text{max}} = \rho + H$ [BN08]
- y axis - the height above the surface normalized by the total height of the atmosphere is used

Both parameters are in the range of [0,1]. Looking at the formula that specifies the parameterization of the coordinate x, the range [0,1] can be easily deduced as we use the min and max values in the calculation. Similarly, the y parameterization range is apparent from the formula, as there is no point in storing values for heights that are outside the atmosphere.

3.1.2 Multiple Scattering LUT

As described previously in Section 2.6, in this work a new approach proposed by [Hil20] is used. The value of Ψ (computed by Equation 2.23), representing the contribution of isotropic multiplescattering to luminance, is stored in a 2D LUT. The parameterization is very similar to the Transmittance LUT:

- x $= 0.5 + 0.5\cos(\theta)$
- y $= \frac{h - R_{Ground}}{R_{Top} - RGround}$

3.1.3 Sky-View LUT

As noted by [Hil20] the overall frequency of the image of the rendered atmosphere is very low. Because of this, it is possible to render the far sky into

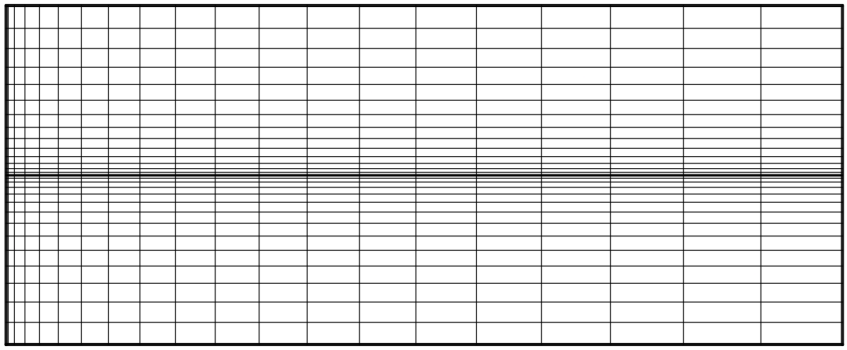


Figure 3.2: Visualization of the skyView mapping where sun moves up and down along the upper half of y range but is always on the left border of the image and horizon is always a dividing line down the middle of the image

a latitude/longitude texture that is of much lower resolution than the final image. It is possible to get good looking results with Sky-View LUT sizes in orders of 10 times lower than the final resolution.

Because the highest visual frequency is introduced by the Sun (mostly) near the horizon, linear parameterization of this LUT would produce artifacts in those high-frequency areas. To solve this Sky View LUT is always oriented in a way such that the sun is present at the same position in the texture. This then gives the option to map the values non-linearly by adding more samples to the areas of high change in frequency.

- x axis - A quadratic curve is used (longitude lon) $x = \left(\frac{lon}{\pi}\right)^2$ giving more samples to the left side of the image where the sun is positioned
- y axis - A quadratic curve is used (latitude lat) $y = 0.5 + 0.5 \cdot sign(lat) \cdot \left(\frac{|lat|}{\pi/2}\right)^2$

where lon is in the range [-π/2, π/2] and lat is in the range [0,π]. Because we are modeling the planet with the assumption that it is perfectly spherical, the latitude range of [0, π] is identical to [0, -π]. This is also the reason why in Figure 3.2 only half of the Sun is mapped into the target texture.

3.1.4 Aerial Perspective LUT

The Aerial perspective refers to how we see objects as they recede into the distance from the viewpoint. This effect can be seen, for example, when looking at distant hills. As the distance of the hills from the viewing position increases, the contrast between the hills ant their background is reduced along with the contrast of any landmarks.

To simulate this effect, a 3D lookup table is precomputed. To parameterize along the z-axis, we use the distance from the viewing position. At each depth level, a 2D LUT is fitted to the camera view frustum. Each layer of

the Aerial Perspective LUT contains the atmosphere luminance and average transmittance at the corresponding depth. The computation of each layer is almost identical to the Sky View LUT. The only difference being that we do not ray march all the way to the edge of the atmosphere but stop at the depth of the corresponding slice. This LUT is then applied as the last step in the sky and cloud rendering process.

3.2 Cloud Precomputations

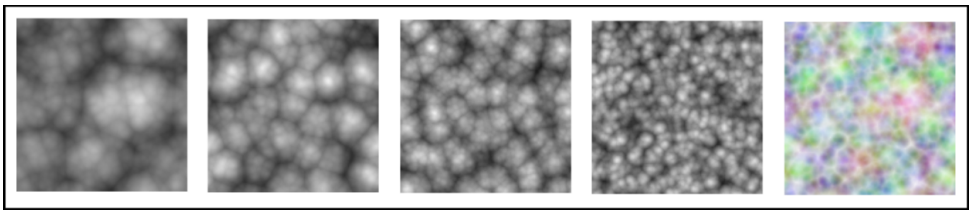


Figure 3.3: From left to right separate RGBA channels storing worley noise and all of the channels combined together in the right most image

As already stated, the very popular method of rendering clouds by [SV15], which this work also uses, relies on the usage of inverted Worley noise. Computation of non-inverted Worley noise can be split into two parts. First, a number of points is randomly distributed in a desired area resp. volume for 2D resp. 3D texture. After this, for each texel resp. voxel in the desired area, the distance to the nearest point was calculated and stored. Inverting Worley noise simply consists of storing $max\_distance - distance$ where $max\_distance$ is the maximum possible distance between a point and a texel, respectively. voxel and $distance$ is the distance from currently processed texel resp. voxel towards the nearest point.

One possible approach could be to scatter those points at random throughout the atmosphere and then compute the nearest distance during the raymarch of the cloud. Although this approach would work, it would be very slow to compute. To decrease the computation time, a technique proposed by [SV15] uses multiple 3D textures that have precomputed Worley noise with various frequencies stored in them. Those textures are then sampled during the raymarch of the cloud, providing a significant increase in speed.

The method implemented in this work is based on the implementation described by [Lag19]. It uses two 4-channel 16bit float textures. Both textures store separate Worley noises with increasing frequencies in each of the RGBA channels. The red channel then stores Worley noise with the lowest frequency and the alpha channel stores noise with the highest frequency.

The texture will have to be tiled multiple times to cover the entire skydome. This gives another requirement for the texture to be tileable (seamless) along all three dimensions.

Chapter 4

Implementation

This chapter describes how the LUTs described in previous chapters are computed and used to produce the final image. As in most performance-dependent applications, this work was implemented using C++. Vulkan API was used as an interface with the GPU. Multiple factors played a role in deciding on the graphics API that would be used. One of the goals of this work was to provide a multiplatform solution (targeting Linux and Windows), which ruled out the use of DirectX almost immediately leaving only OpenGL and Vulkan. While OpenGL is more friendly in terms of the API, it was decided to instead use Vulkan and pay the price of more upfront work (and LOC) in setting all of the things handled by OpenGL's driver side and in return get the performance benefits of more manual control over GPU exposed by Vulkan.

4.1 Application resources

In this section, we describe all the application resources and their format. We will mostly omit small uniform and storage buffers used only for parameterization, as they are multiple orders smaller than the LUT textures and have no real effect on the memory requirements of the application. Please note that, if not mentioned otherwise, each resource is allocated once for each *frame in flight* we want to have.

4.1.1 Atmosphere and clouds

When rendering the atmosphere four previously described LUTs have to be computed. We use 16 bit RGBA texture for each LUT. The parameterization that we decided to use can be seen in Table 4.1.

Look up table	Resolution	size in bytes
Transmittance LUT	256 × 64	128 KiB
Multiscattering LUT	32 × 32	8 KiB
Sky View LUT	192 × 128	198 KiB
Aerial Perspective LUT	32 × 32 × 32	256 KiB
Total		590 KiB

Table 4.1: Parameterization and sizes of LUT's used to render atmosphere.

In addition to this we use two volumetric textures used to store Worley noise. Similarly to LUT's used to render atmosphere, these textures store 16 bit floating point values in each of the channels. The parameterization along with the size can be seen in Table 4.2. The resolution of the noise textures is slightly higher than described by [SV15]. We felt like it gave us higher quality results with less visible tiling, but it could easily be lowered to half or more and still produce good looking results.

Look up table	Resolution	size in bytes
Base Shape Noise LUT	256 × 256 × 256	128 MiB
Detail Noise LUT	128 × 128 × 128	16 MiB
Total		144 MiB

Table 4.2: Parametrization and sizes of LUT's used to store Worley noise.

4.1.2 Sky

Because our implementation uses HDR as it's preferred format, we need a texture capable of storing HDR values throughout the process of rendering a frame. For this purpose we use one 32bit floating-point RGBA texture with the same resolution as swap chain images. We also make use of two depth buffers during each pass. Because we make no use of stencil buffer, we decided to use one channel 32bit depth format for each depth texture. The memory requirements for both 720p and 1080p resolution can be seen in Table

Texture	Resolution	size in bytes
720p HDR Backbuffer	1280 × 720	14.1 MiB
720 Depth buffers	1280 × 720 × 2	7.05 MiB
Total		21.15 MiB
1080p HDR Backbuffer	1920 × 1080	31.65 MiB
1080 Depth buffers	1920 × 1080 × 2	15.82 MiB
Total		47.5 MiB

Table 4.3: Parameterization and sizes of HDR backbuffers

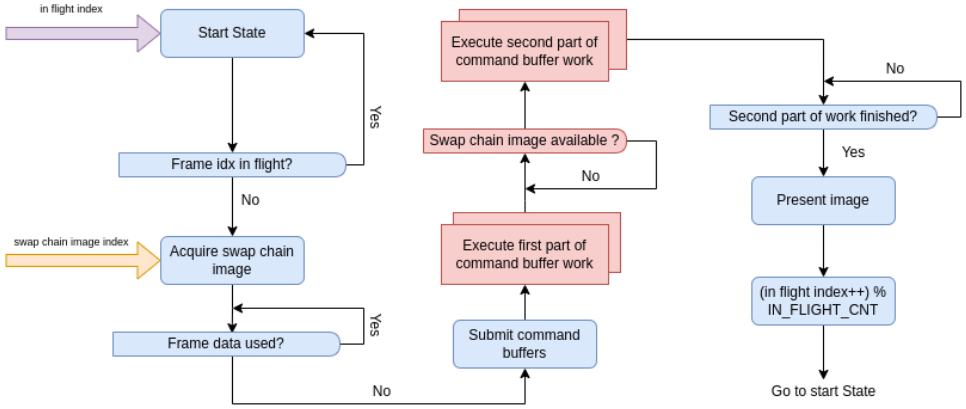


Figure 4.1: Flow diagram of the draw loop execution order. CPU parts as well as CPU-GPU synchronization points are colored blue. Similarly GPU parts and GPU-GPU synchronization are colored red

4.2 Main Draw Loop

Because most of the command buffers in our implementation are pre-recorded and do not need rebuilding, the main purpose of the draw loop is to keep the GPU fed with as much work as possible. In order to do this, Vulkan allows us to submit work for multiple frames in advance, which means that we can render a new frame while the previous frame is still in the process of rendering. We will refer to this concept as having multiple *frames in flight*. By default, three frames are in flight in our implementation (which is incidentally also the amount of swap chain images we create).

There are multiple caveats to having multiple frames in flight. First, we have to ensure that the work submitted to the GPU is actually finished in time. If we do not do this and the GPU is slower in executing the command buffers than the CPU is in submitting them, the corresponding command queues will fill slowly and eventually overflow. Another thing we need to avoid is two (or more) frames in flight that are modifying this resource in any way. modified during the rendering process. To avoid this, multiple GPU-GPU and CPU-GPU synchronization points are implemented in a similar fashion as described in [Ove21].

At the start of our loop, we check if we do not already have more images in flight than we want. For this purpose, we create a fence for each frame in flight that we want to have. When we are sure that the number of frames in flight is less than the maximum specified values, we continue by acquiring the next image index from the swap chain. This prevents the above-mentioned issue of slowly overflowing our command queues.

If we specified more maximum frames in flight than there are swap-chain images or if the swap-chain returned image indexes out of order, the previous fence would not stop us from ordering the GPU from using the same set of

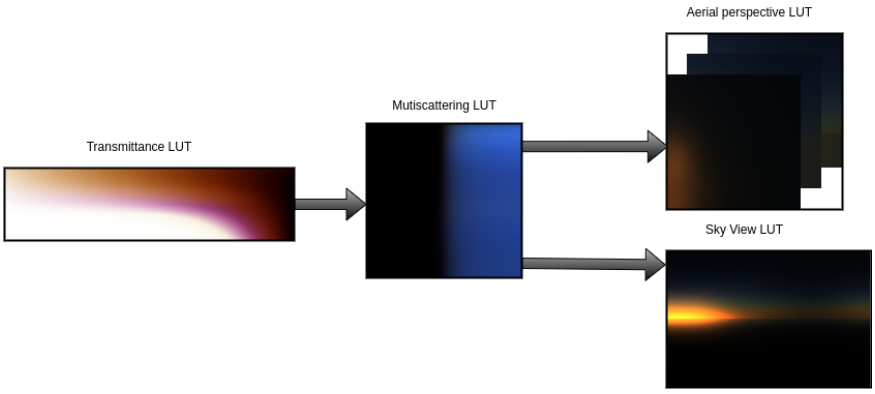


Figure 4.2: The order in which individual Sky Luts are drawn. Please note that color values of LUTs have been scaled in order to be properly visible

data for multiple frames. Because of this, we have an array of structures containing all of the data that change during the process of rendering one frame and a fence specifying whether the data are currently being used by some in flight frame. Whenever a new image is acquired from the swap chain it checks the corresponding frame data structure fence and only after the fence has been signaled an appropriate command buffer is submitted to GPU. Whenever we finish rendering of any frame, a structure fence is signaled, allowing another frame to be submitted. Figure 4.1 shows a flow diagram visualizing the entire draw loop.

For each frame, four command buffers are submitted to the GPU. The first pair of command buffers can start executing immediately, but since the second pair of command buffers writes directly into a swap chain image, we need to make sure the corresponding image is available for us to write. We use an additional array of semaphores. Each semaphore is signaled when the presentation engine is finished using the corresponding image.

4.3 Command buffer descriptions

In this section, we will provide a fairly detailed description of the commands that are submitted in each command buffer. We will also describe the GPU-GPU synchronization that takes place inside each of the command buffers. Think of this section as a description of Vulkan specific parts in our implementation. This, of course, is not everything that is Vulkan specific in our application; however, as we did not believe those other parts unique to our implementation, we decided to omit them.

4.3.1 LUTs command buffer

When pre-computing LUT entries most of the graphics pipeline's features would get in our way instead of helping us. Due to this in our implementation, we decided to use compute shaders to fill out all LUTs. Each LUT is computed

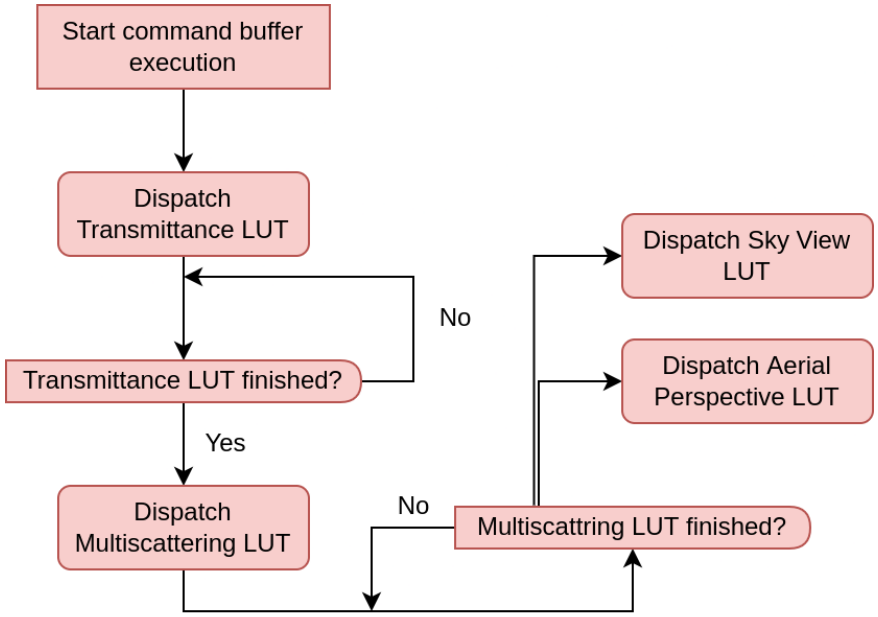


Figure 4.3: Flow diagram showing dependencies between individual operations in LUTs command buffer. As all parts are either executed on GPU or are GPU-GPU synchronization they are all marked red

by one shader in one dispatch.

Compute dispatch commands are recorded in the order shown in Figure 4.2 in the command buffer. Because the intermediate results of Transmittance LUT are needed when filling Multiscattering LUT and similarly intermediate results of Multiscattering LUT are needed when calculating SkyView and Aerial perspective LUTs we need to introduce some sort of synchronization between the individual dispatch commands. We decided to use pipeline barriers for this. This results in us waiting, after each drawcall, utill all of the previously submitted compute work has been finished before issuing another dispatch. Figure 4.3 shows the visualization of the execution order in this command buffer, as well as the synchronization performed between executions.

4.3.2 Worley Noise command buffer

As described in Section 4.2 sometimes an additional LUT command buffer may be submitted. This command is used to compute the 3D Worley noise texture used in later phases. Similarly to the previous section we have no use for graphics pipeline's features when generating this texture, so we use compute shaders here as well.

We decided to follow the approach used by [Lag19] and render the texture in two separate passes. The first pass renders the Worley noise itself, and the second pass normalizes the values to the range between 0 and 1. Since each channel of the resulting 3D texture stores completely separate Worley noise, we decided to use a separate one-channel texture for each of the final

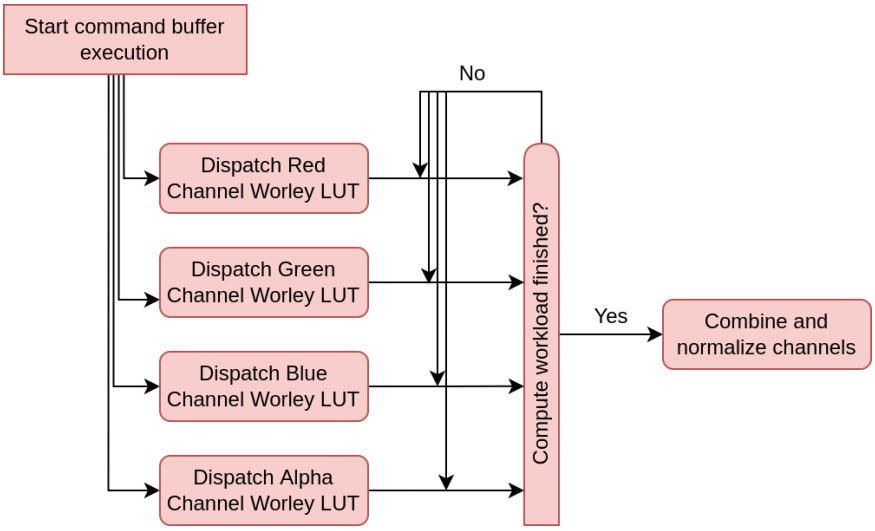


Figure 4.4: Flow diagram showing dependencies between and execution order of worley noise command buffer

four textures in the first pass. We use these four textures as input to the second pass, which normalizes and combines them by storing values from each texture in the corresponding channel.

As far as we know, it is not possible to write only one channel of a texture at a time from compute shaders. For this reason if we only used a single texture each dispatch would have to read the value previously stored in all of it's channels, modify single channel, and write back values for all four of these channels. In addition to loading unused data from VRAM we would also have to introduce synchronization between dispatches writing into separate channels to avoid write after write hazards etc...

By splitting the texture into four separate textures in the first pass, we avoid all of the problems described above. We don't need any kind of synchronization between the writes to a separate texture. Only synchronization that we must ensure happens is the one between the first and second pass. To ensure this, a single pipeline barrier is inserted making sure all writes and reads in compute shader stage by previously called dispatches have finished before we normalize and combine all channels together. The entire execution process can be seen in Figure 4.4

4.3.3 Sky command buffer

When all LUTs have been precomputed, we can move on to rendering all the objects we want in our scene and drawing the sky (atmosphere) with clouds. The ordering here is important; we first need to render all the objects before drawing the sky and clouds. This is because drawing sky, clouds, and atmosphere requires depth information about the rest of the scene.

As mentioned above after all scene objects were rendered far sky is drawn to the sections of the image which have maximum depth value still stored,

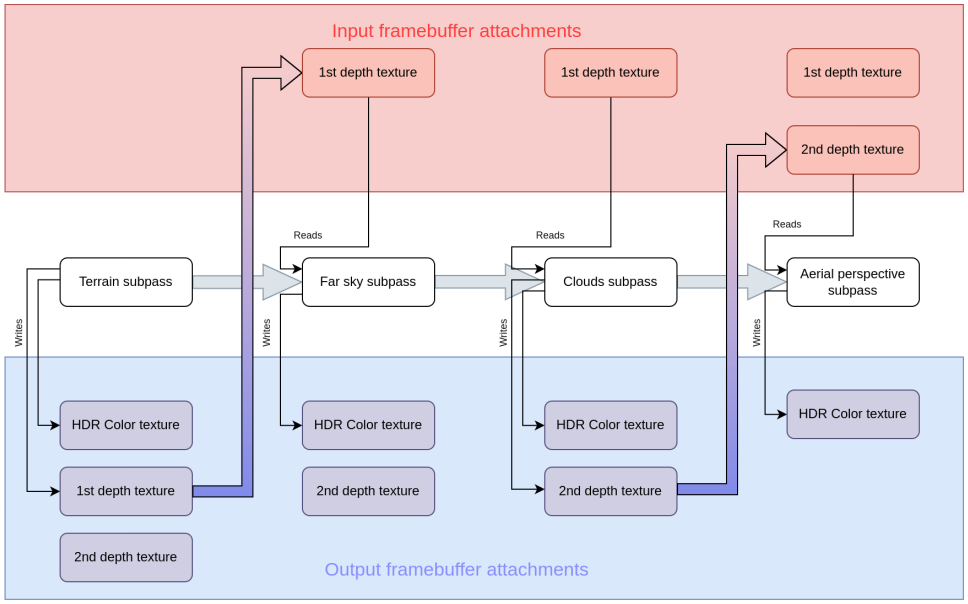


Figure 4.5: Visualization of writes and reads performed by each subpass. Additionally we also show when do transitions from output attachment into input attachment inside a framebuffer occur

meaning no object is occluding the view. After this, the clouds are rendered. Whenever scene object is only partially occluded by cloud (as an example we can imagine tops of mountains poking into the bottom of the cloud layer), we don't want the object to fully disappear. In order to properly display this effect, we will require a scene depth buffer as well. Lastly, the aerial perspective is applied. Recall that in Section 3.1.4 we define the aerial perspective as a reference to how we see objects as they recede into the distance. With this definition in mind, it is now clear why we need scene depth information during this rendering phase as well.

There is one problem. We would like to apply aerial perspective coherently to the entire scene. Clouds, however, are part of the scene, so we want to draw the effects of aerial perspective on top of the clouds as well. To do this, we need information about how far the clouds are from the viewing point stored in the depth texture. But above we mentioned that clouds need depth texture as input as well. This would mean that the pass rendering clouds wants to read and write the same texture. To avoid this problem, we introduce a second depth texture. When rendering clouds, we then read from the first texture which contains depth values of scene objects combine with depth values of the clouds and write the result into the second texture. The second texture is then used as input into the aerial perspective pass. Figure 4.5 shows how every subpass reads or writes resources from the framebuffer.

Similarly to LUT computation, we need to introduce GPU-GPU synchronization between individual draw calls to make sure the previous part of the rendering has finished. However, we can be less strict in this case. Because we know that in each pass of the sky rendering process we will be accessing

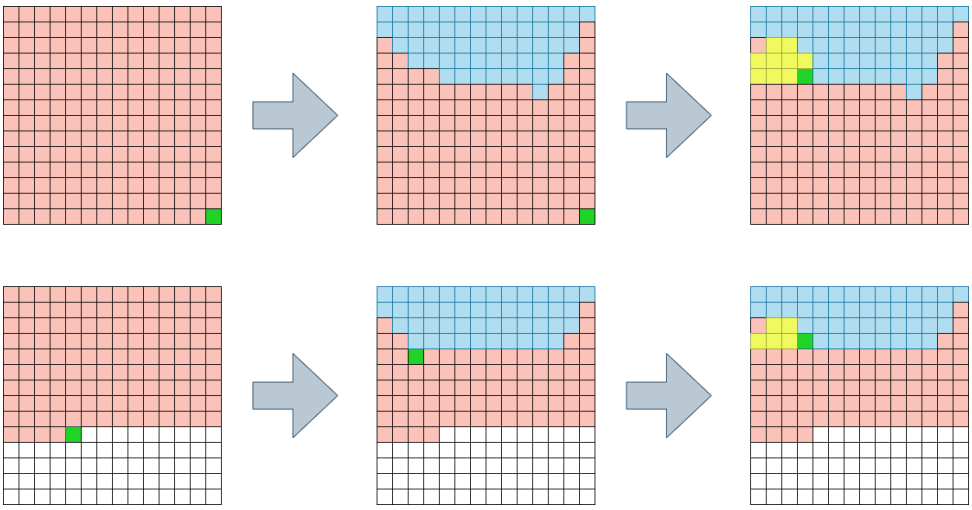


Figure 4.6: The top sequence shows traditional style of rendering, where we wait to render the entire image before continuing with next pass. Fragment highlighted in green shows which position current dispatch is shading. We can see that only third pass is active and previous passes already finished (visualized by active fragment being in the low right corner). Bottom sequence shows how local dependencies could render image. We can see that all three passes are rendering at the same time but only on the fragments which were already processes by previous passes

the same *framebuffer fragment region* (think fragment), we do not need to wait for the previous pass to finish the entire draw workload. We can instead just make sure that the previous pass has completed all the writes or reads of the *framebuffer fragment region* we are currently processing. This gives us the option to, for example, start drawing clouds before the entire far sky has been rendered. We only instruct the GPU to make sure that all operations performed by previous passes above the current *framebuffer fragment region* are finished before the current pass reads or writes any values.

Note that we cannot make the same assumption when computing LUTs. There are multiple reasons for this, but the main ones are as follows:

- 1. We have no way of knowing which LUT regions will be accessed by the following stage
- 2. We use different texture types (2D and 3D) as well as sizes for each LUT so we are unable effectively unify them into a single framebuffer.

Because the entire sky-draw process reads and writes the same three textures, we do not need to specify a separate framebuffer for each pass. The only thing that changes between the passes is how the framebuffer textures are used. Luckily for us, this specific case is common enough that Vulkan exposes certain parts of it's API just for this purpose, which are called subpasses and subpass local dependencies.

A subpass represents single phase of rendering that reads and writes a subset of the attachments in a render pass as well as describing the layout transitions

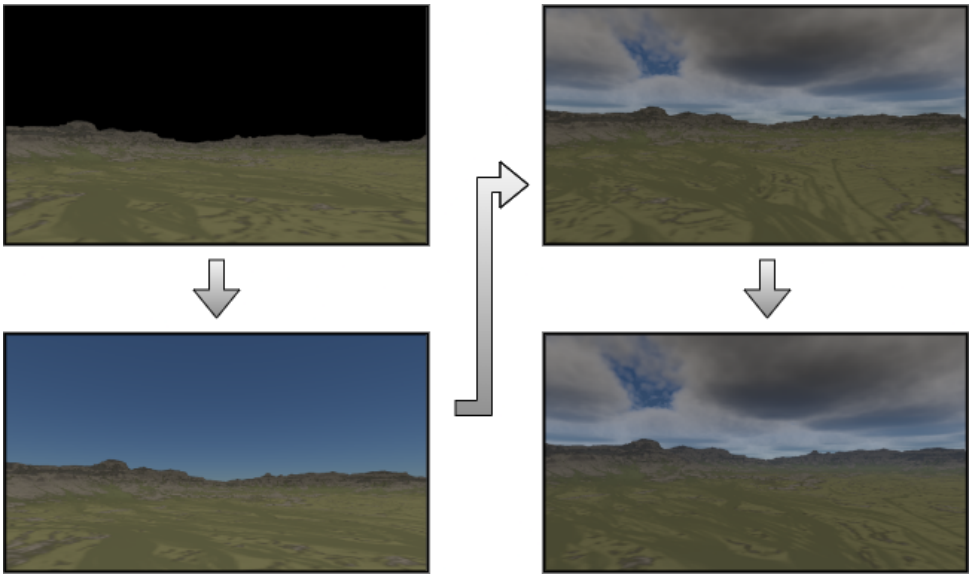


Figure 4.7: This image shows how the image changes in each of subpass stages.

and dependencies between multiple subpasses. A subpass dependency is essentially an automatically inserted pipeline barrier at the start or end of each subpass. A subpass local dependency (or dependency by region) is a type of subpass dependency that is only framebuffer local. A framebuffer-local dependency is exactly what we described above, where separate subpasses only access the same *framebuffer fragment region*.

We use a single render pass to render the sky. This render pass is divided into four sub-passes. Each subpass is responsible for one of the previously described phases in the described order:

1. subpass - draw all scene objects
2. subpass - draw far sky
3. subpass - draw clouds
4. subpass - draw aerial perspective effects

Local dependencies between each subpass are specified as follows. First, since every subpass writes into the same HDR framebuffer attachment, we need to make sure to avoid write after write hazard. This can be solved by inserting a local dependency guarding writes to a color attachment.

In addition, far sky and cloud supbasses read from the first depth attachment written by scene objects subpass. Similarly, aerial perspective subpass reads from the second depth attachment written by clouds subpass. To prevent a read after write hazard, additional subpass dependencies are added, which make the start of fragment shader in a subpass wait before early and late fragment tests (this is where all writes into a depth buffer occur) were finished by the previous subpasses. In addition, specified subpass dependen-

cies also take care of the attachment transitions from input attachments to output, etc.. . . where needed.

4.3.4 Post process and GUI command buffers

Final two buffers submitted by the application each frame are buffers that contain the post-processing of the final image and the drawing of the UI. Post processing the image consists of tonemapping luminance values stored in HDR buffer into LDR range and gamma correction. For the purposes of tonemapping, we first need to calculate the average luminance of the current image. We use a two-pass compute approach described by [Tar19].

In the first pass, a histogram of the luminance values in the image is constructed. The second pass reads this histogram and calculates the weighed sum. This sum is then used to calculate the adaptive average luminance of the scene. Our post process fragment shader then reads this average value and uses it for tonemapping.

The synchronization used in this command buffer is very similar to the one used during the LUT computation. A pipeline barrier is inserted between constructing the histogram and calculating the average luminance, making sure that all compute stage writes have finished before reading anything. A second pipeline barrier is inserted before tonemapping draw call to make sure the previous dispatch is done writing the average luminance.

The last command buffer we will describe is the one that shows all the UI on the screen. It does not need any kind of synchronization done by us, as everything is handled internally by the ImGUI implementation.

4.4 Shaders

In this section, we describe how we implemented most of the shaders used in our application. Although we tried to provide a pretty detailed description, we still had to omit some less important or interesting parts. The complete code for all shaders can be found here or in the attachment. Shaders are written in GLSL. They are compiled along with the application into SPIR-V intermediate format and are loaded as a part of the initialization sequence of the application.

4.4.1 Raymarch implementation

Ray marching is a key technique that many of our shaders use. Due to the frequency of it's appearance in our implementation, we decided to dedicate a separate section to explain it. This makes it so that we can later just refer to this subsection instead of duplicating it throughout the rest of the shader implementation section. Whenever we use the function call `raymarch(...)` or `raymarch_nonuniform(...)` we refer to this section.

As hinted at in the previous paragraph, we make use of two different ray march functions. The difference between those functions is the distribution of

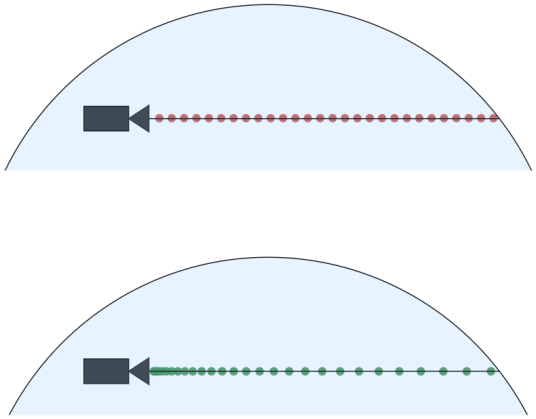


Figure 4.8: In the top part of the image we can see linear sampling with constant distribution. Bottom part shows non-linear sampling with reducing sample density based on the distance from the ray origin

ray march samples along the ray. The first function samples the path linearly, always moving by a constant offset. The second nonuniform function, as the name suggests, distributes the samples nonuniformally with a higher density of samples near the ray origin. The further we go from the origin of the ray, the bigger steps we take along the ray. This can, in some situations, lead to better looking images and generally nicer results.

Listing 4.1: Linear raymarch function

```
// position of last step is stored here
vec3 oldPos = vec3(0.0, 0.0, 0.0);
vec4 raymarch(vec3 dir, vec3 pos, float iter,
    float iterCnt, float rayLength)
{
    // Sample at one third of interval - gives better
    // results when integrating exponentional functions
    float rayShift = rayLength * (iter + 0.3) / iterCnt;

    vec3 newPos = pos + rayShift * dir;
    float dt = newPos - oldPos;
    oldPos = newPos;
}
```

In Listing 4.1 we can see our implementation of the linear ray march function. Parameters `dir` and `pos` denote the starting position and direction of the ray. Following this `iter` and `iterCnt` are the current step index and the amount of steps our ray will be divided into. Lastly, `rayLength` is the distance over which we want to distribute our samples.

Listing 4.2: Nonlinear ray march function

```
vec4 raymarch_nonuniform(vec3 dir, vec3 pos, float iter,
    float iterCnt, float rayLength)
{
    step0 = pow(iter / iterCnt, 2);
    step1 = pow(iter + 1.0 / iterCnt, 2);

    realStep0 = step0 * raymarchLength;
    // Make sure not to overshoot the raymarching distance
    // in the last sample
    realStep1 = max(step1 * rayLength, rayLength);
    float rayShift = realStep0 + (realStep1 + realStep) * 0.3;

    vec3 newPos = pos + rayShift * dir;
    float dt = realStep1 - realStep0;

    return vec4(newPos, dt);
}
```

In Listing 4.2 we can see the extra step of squaring the step size first, condensing more samples towards the origin position.

4.4.2 Atmosphere and Sky LUT computation

Unless stated otherwise, all shaders described in this section are largely based on the open source code provided by [Hil20]. However, it is important to note that the shaders provided by Hillaire were written in HLSL. This meant that before using any shader, we had to port it to GLSL first. An additional effort was made to refactor most of the code and provide slightly more comments.

Transmittance

Our transmittance LUT implementation is based on a freely available source provided by [BN08]. For each LUT entry, we ray march a single ray from a specified direction towards the end of the atmosphere. We use the linear `raymarch()` function for this purpose. During this, we sample the density of the atmosphere from which we calculate the extinction coefficient and transmittance exactly as described in Section 3.1.1.

The only change we have made is that instead of ray marching three times separately for *rayleigh extinction*, *mie extinction* and *ozone extinction* we ray march only once sampling all three values at the same time, thus reducing the computation time to almost one third.

Multiscattering

As described in previous chapters, multiscattering LUT samples rays uniformly distributed along a sphere. We decided to follow the example provided

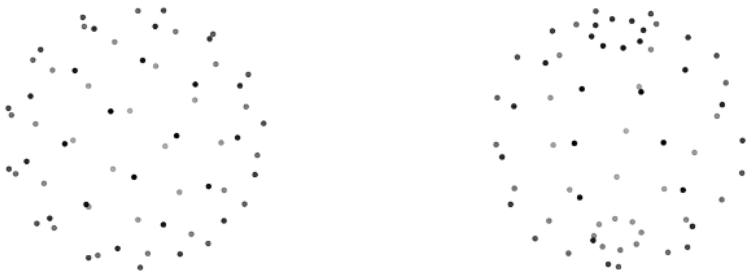


Figure 4.9: On the left distributing points using Fibonacci lattice. On the right distribution Hillaire uses in his implementation. Bias toward top and bottom of the sphere is clearly visible

by [Hil20] and use a local team of 64 threads for this purpose. Each local thread is assigned a ray direction based on its index, and the ray marches (using `raymarch(...)`) until it hits the ground or the end of the atmosphere. During each step, we sample the scattering and extinction of the medium and calculate $L_{2^{nd}order}$ and f_{ms} as described in Equations 2.19 and 2.20.

Sampling the sphere. For distributing samples along a sphere, we used spherical coordinates. Previous solution distributed samples using uniformly increasing angles θ and ϕ. After this they were converted back to Cartesian coordinates by using a standard equation:

$$\begin{aligned} x &= \cos(\theta) * \sin(\phi) \\ y &= \sin(\theta) * \sin(\phi) \\ z &= \cos(\phi) \end{aligned} \tag{4.1}$$

We decided to use a slightly better distribution along a sphere using *Fibonacci lattice* described by [Gon09]. Points are distributed along the sphere by the following equations:

$$\begin{aligned} \theta &= \arccos\left(1 - \frac{2n+1}{N}\right) \\ \phi &= \frac{2\pi n}{\Phi} \end{aligned} \tag{4.2}$$

Where Φ is the golden ration defined as $\frac{1+\sqrt{5}}{2}$. This produced samples that are less biased toward the top and bottom of the sphere, which can be seen in Figure 4.9. The effects of our sampling are compared to Hillaire's sampling in Figure 4.10. The differences are most visible in the LUTs generated with a half-particle density. Reduced bias in sampling results in brighter values when evaluating samples near the ground (top of the image). When using higher scattering coefficients, the effect becomes very small, resulting in almost identical values.

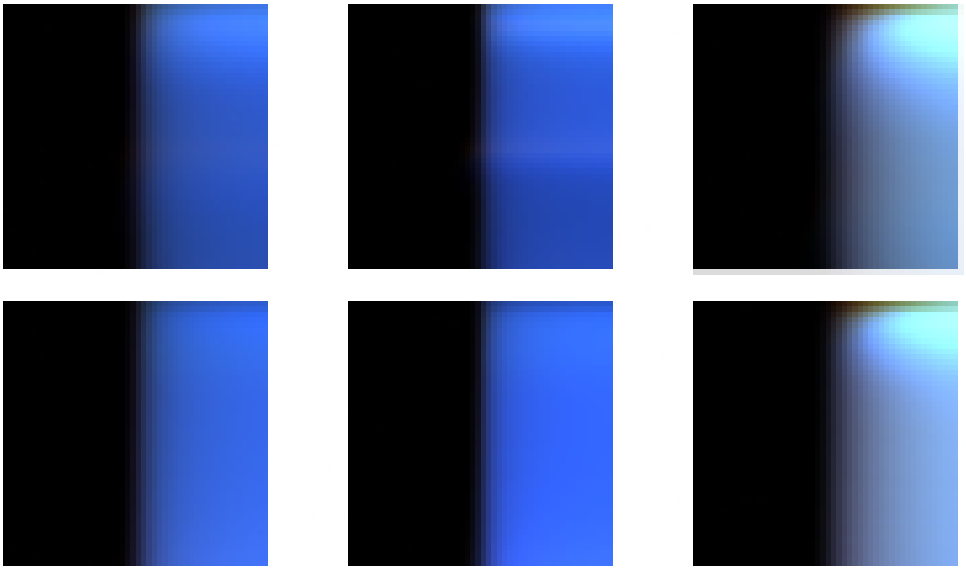


Figure 4.10: The effects of sampling methods on Multiscattering LUT. Top row shows results when using Fibonacci lattice. Bottom row shows results of sampling used by Hillaire. From left to right Earth-like atmosphere, atmosphere with half of the particle density and atmosphere with 10x increased Rayleigh and Mie scattering coefficients. Colors were scaled for the results to be visible by respective factors of 50, 20 and 100.

Merging the results. After every thread in the local workgroup finishes the ray marching loop, we need to merge the results into a single value to be able to evaluate Equation 2.23. In order to do this, we create two local, group shared arrays the size of our work group (specifically 64 in our implementation). Each thread stores its $L_{2^{nd}order}$ and f_{ms} values into those arrays using their local group ID as an index whenever it finishes ray marching.

Following this we use thread based merge to combine all of the results into two final values. We then use those values to calculate Equation 2.23 and store this result. We can also see the code for this in Listing 4.3.

Listing 4.3: Multiscattering results merge

```
1                 ...
2 // raymarch finished merging results (extra barrier
3 // is before this to ensure raymarch finish)
4 uint threadIdx = gl_localInvocationID.z;
5 if(threadIdx < 32)
6 {
7     f_ms_shared[threadIdx] += f_ms_shared[threadIdx + 32];
8     L_shared[threadIdx] += L_shared[threadIdx + 32];
9 }
10 // Make sure previous part of merge finished
11 groupMemoryBarrier();
```

```
barrier();
if(threadIdx < 16)
{
    f_ms_shared[threadIdx] += f_ms_shared[threadIdx + 16];
    L_shared[threadIdx] += L_shared[threadIdx + 16];
}

...

// we only want sigle thread past this point
if(threadIdx != 0) {return;}
//calculating Equation 2.23
vec3 res = L_shared[0] * vec3(1.0 / 1.0 - f_ms_shared[0]);
imageStore(multiscatteringLUT, uv, vec4(res, 1.0));
```

SkyView

Calculating Sky View LUT is relatively simple. First, we transfer from uv coordinates in the range [0, 1] to LUT parameters described in Subsection 3.1.3. Because our ray march function uses Cartesian coordinates we first need to convert from spherical coordinates (used by Sky View LUT parameterization). We use the same conversion as described in Equation 4.1 giving us the direction of ray that we need to ray march.

Listing 4.4: Multiscattering results merge

```
#define ITER_CNT 30;
void main()
{
    // Convert global invocation idx in range
    // [0,skyViewRes] into uv range [0, 1]
    vec2 uv = gl_GlobalInvocation.xy / skyViewLUTDim;

    // camera pos in UBO offset by groud radius
    vec3 pos = getPos();
    vec3 dir = lutToDir(getLUTParams(uv));

    // get the ray intersection with the atmosphere
    float rayLen = getRayInt(pos,dir,atmoBounds);

    vec3 accumT = vec3(1.0);
    vec3 accumL = vec3(0.0);
    for(int i = 0; i < ITER_CNT; i++)
    {
        vec4 res = raymarch_nonuniform(dir, pos, i,
            ITER_CNT, rayLen);

```

```
        // res.xyz stores new pos, res.w stores step size
        float transIncRay = exp(-(medium.ext * res.w);
        vec3 light = calcLight(res.xyz, transIncRay);

        accumL += accumT * light;
        accumT *= transIncRay;
    }
    // Store accumL into skyViewLUT
    ...
}
```

Using this together with the current camera position in the world space, we ray march the full rendering Equation 2.16. In order to produce better results, we use `raymarch_nonuniform(...)`. Please note that we make use of the previously calculated Multiscattering LUT to evaluate orders of scattering higher than one.

A shortened version of the source code can be seen in Listing 4.4.

Aerial perspective

Aerial Perspective LUT uses essentially the same computation as Sky View LUT, with few slight changes. First, because individual Aerial Perspective LUT layers are fitted on the view frustum in appropriate depth, we need to project from screen space coordinates into coordinates to get direction of the ray we want to ray march. The listing 4.5 shows the code we use to achieve this. We first covert the screen coordinates into NDC coordinates. We the use the inverse of View-Projection matrix and divide by the new w component to get the position in the world space. To get the view direction, we take the vector from the camera to our newly acquired point and normalize it.

We also limit the ray length to not be longer than the distance of the currently computed slice. Using those slightly modified parameters, we execute the exact same ray march as used for the Sky View LUT computation. Lastly, we also store the average transmittance (averaged of all three wavelengths) in Alpha channel of each LUT layer.

Listing 4.5: Function transfer from screen coordinates into world direction

```
vec3 screenToWorldDir(vec2 uv, vec3 pos)
{
    vec3 NDC = vec3(uv * vec2(2.0) - vec2(1.0), 0.5);
    vec4 Hpos = invViewProjMat * vec4(NDC, 1.0);
    vec3 dir = normalize(Hpos.xyz / Hpos.w - pos);

    return dir;
}

```

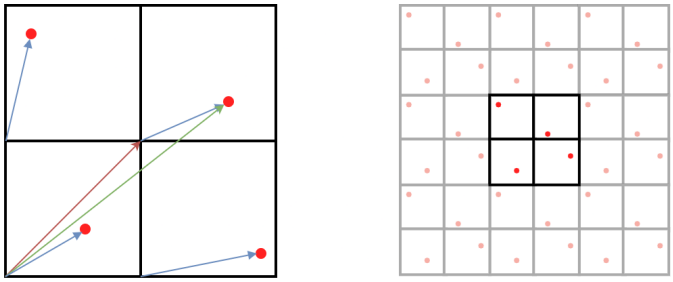


Figure 4.11: On the left 2D unit volume with one division along each axis. Red arrow denotes cell offset, blue arrows denote cell local random offset and green denotes the global position of a point. On the right the same 2D mirrored volume setup which produces seamless texture.

4.4.3 Worley Noise computation

Now we will describe the shaders used for both passes of our 3D inverted Worley noise generation adapted from [Lag19]. We use three octaves of noise with increasing frequencies, which are then combined into a single noise using the standard Fractal Brownian Motion approach. During the generation of each noise texture, we sample a unit cube filled with randomly scattered points. The naive approach of iterating through all points scattered in the medium to get the nearest point is very slow.

To decrease the number of points that we need to check, we divide the unit cube uniformly into a set of smaller cells, each containing a single point. By doing this, we guarantee that the nearest point will lie either in the cell containing evaluated voxel or in one of its surrounding cells. For each voxel, we can then reduce the number of checked points to only 28. To increase the frequency, we simply divide the unit volume into more cells.

To distribute the points randomly, we first generate an offset vector for each point in the local space of the cell. This offset is then translated from the local cell space into the unit range global space by summing it with the cell origin (front-bottom-left corner of the cell). Point generation is done on the CPU, after which we pass the point array to shaders as SSBO.

The increasing frequency is achieved by dividing the unit range into more cells. We generate three buffers of points, one for each octave of noise. To fulfill the seamless constraint, we must mirror the sampled volume and attach it to each side of our sampled volume. This, along with the local-to-global cell mapping, can be seen in Figure 4.11. The point data are actually not duplicated; whenever we are near the edge, we just recalculate the neighbor index to the correct mirrored cell.

Listing 4.6: Worley noise computation

```
1 void main()
2 {
3     vec3 index = vec3(gl_GlobalInvocationID.xyz);
4     vec3 pixPos = index/texDimensions;
```

```
5       // layer A, B and C corresponds to 1st, 2nd
6       // and 3rd octave of noise
7       float layerA = worley(0, pixPos);
8       float layerB = worley(1, pixPos);
9       float layerC = worley(2, pixPos);
10
11      // combine all three octaves
12      float combinedNoise = layerA;
13      combinedNoise += layerB * amplitude;
14      combinedNoise += layerC * amplitude * amplitude;
15
16      float localMax = 1 + amplitude +
17            amplitude * amplitude;
18      // make sure the noise values are in the range [0,1]
19      // and invert them
20      combinedNoise /= localMax;
21      combinedNoise = 1.0 - combinedNoise;
22      atomicMin(minVal, combinedNoise);
23      atomicMax(maxVal, combinedNoise);
24
25      imageStore(...);
26  }
```

The code used to calculate one Worley noise texture can then be seen in Listing 4.6. One thing we have not described yet is the pair of atomicMin and atomicMax on lines 22 and 23. Because we are always sampling unit cube, the more divisions we have, the smaller the distances between individual points will be. After inverting the values, this results in values that do not cover the entire range from zero to one.

As part of the second pass, we want to correct for this by remapping the values. However, in order for us to do so, we need to know the max and min values which occur in the texture. This is exactly what the code on lines 22-23 does, stores minimum and maximum values, which were written into the texture in another small SSBO.

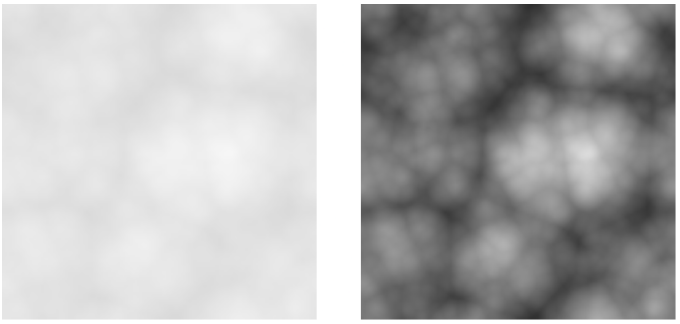


Figure 4.12: On the left raw values before normalization, on the right values after normalization.

The only thing that second pass then does is combine the four single-channel textures into a single four-channel after normalizing the values. The code for this shader is presented in the Listing 4.7, the effect of normalization of the noise can be seen in Figure

Listing 4.7: Worley noise normalization and combine pass

```
void main()
{
    float resultArr [CHANNEL_CNT];
    for(int i = 0; i < CHANNEL_CNT; i++)
    {
        float noise = loadVal(uv, i);
        float normNoise = (noise - minVal[i])/
                          (maxVal[i] - minVal[i]);
        resultArr[i] = normNoise;
    }
    vec4 result = vec4(resultArr[0],resultArr[1] ... );
    // Store the result in final texture
}
```

4.4.4 Frame composition

In this subsection, we will describe all shaders that are used to draw the HDR image that is used later in post processing. We describe the shaders in the order of execution described in 4.3.3. Thus, each shader directly maps to a draw call executed in each subpass. Unless otherwise stated, the shaders described in this section are again adapted from [Hil20] and [Lag19] and modified in some places to fit our application.

Terrain

To draw the terrain, we first generate a subdivided unit length rectangle of points on the CPU. We use these points as input to the vertex shader. For each vertex, we then sample the height map texture and offset the height by the sampled value. Lastly, we scale the model using the model matrix.

In the fragment shader stage, we use *Phong reflection model*. Normals are read from a normal map texture. To get the color of each fragment, a color mask map is used. Each channel of the color-mask map corresponds to the weight of a single color. We multiply each of the four terrain colors by the sampled weight. To get the color of the sun light hitting the terrain we sample Transmittance LUT giving us the colored transmittance towards the Sun. Please note that our Transmittance LUT only stores the transmittance of the atmosphere and does *not* include the transmittance of the cloud layer.

Far sky

The shader for drawing the far sky is also relatively simple. First, we check if the value stored in depth buffer written by previous subpass is equal to the clear color. If it is, we know there are no objects obstructing the view and we can continue by drawing the sky. Otherwise, we do not draw anything and return early.

In order to sample the Far Sky LUT we need to convert the screen space coordinates into latitude and longitude used for Far Sky LUT parameterization. This step is exactly the same as the one used when calculating the Aerial Perspective LUT. In fact, we can think of the Far Sky LUT as a last layer of Aerial Perspective LUT with higher resolution, which is not fitted to the camera frustum but instead fitted to the entire viewing range. Because of this similarity we will not describe the calculation used to transform from screen space to world space and instead refer to Section 4.4.2 where this process is explained.

To get Sky View LUT parameters, we need to know the angle of our ray from the horizon and the angle of our ray to the Sun. To get the angle to the Sun, we project both sun direction and view direction vectors onto the xy plane and calculate their dot product. The angle from the horizon is given by the dot product of the up vector with view direction vector. We can see this computation in Listing 4.8. In the next step, we perform the inverse mapping of Sky View LUT to get the texture UV coordinates. As a last step, we sample the Sky View LUT with our calculated UV coordinates and store the sampled value.

Listing 4.8: Screen position to Sky View LUT UV

```
vec2 posDirToLUTParams(vec3 pos, vec3 dir)
{
    vec3 up = normalize(pos);
    float horizonAngle = acos(dot(dir, up));

    vec3 dirProj = normalize(vec3(pos.xy, 0.0));
    vec3 sunProj = normalize(vec3(sun_dir.xy, 0.0));

    float sunAngle = acos(dot(sunProj, dirProj));
    // Perform Sky View inverse mapping
    return SkyViewParamsToUv(horizonAngle, sunAngle);
}
```

Clouds

The first step we need to take before starting the ray march is to figure out the starting position and the length of the ray we want to ray march. We calculate the intersection distances of the rays with the start of the cloud layer, the end of the cloud layer, and the planet's ground. If the distance to

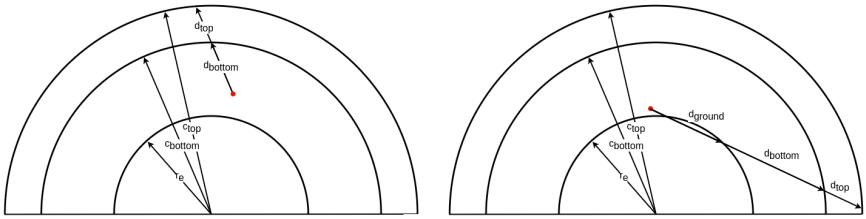


Figure 4.13: Overview of two situations which can arise when figuring ray intersections with cloud layer. r_e is the planet radius, c_{top} and c_{bottom} are the top and bottom of cloud layer. d_{ground}, d_{bottom} and d_{top} are the intersections with planet ground, start of the cloud layer and end of the cloud layer.

the planet ground is greater than zero, we never hit the cloud layer. If, on the other hand, we do not hit ground, the intersection with bottom of the cloud layer is the distance to our cloud layer. We then subtract the distance to the top layer intersection from the bottom layer intersection to get the distance through the layer. We can see this in Figure 4.13.

After this we offset the position of the camera by the distance towards the cloud layer along the viewing ray and start ray marching. The samples are distributed linearly through the medium, so we use the `raymarch(...)` function for this purpose.

To estimate the amount of light reaching each sampled position, we need to perform a secondary ray march towards the Sun. We use only a small number of samples for this purpose because the computation complexity increases very quickly with increasing numbers of secondary samples. The light reaching each sample is then attenuated by the transmittance accumulated along the ray towards the ray origin and the atmosphere transmittance towards the sun stored in Transmittance LUT.

An additional improvement suggested by [SV15] is to use an additional *Powder effect factor* when calculating the transmittance. This effect modifies our approximation slightly. With increasing depth inside a cloud, the probability of light inscattering in the ray direction increases. To model this increase, we multiply our transmittance calculated by an additional *Powder effect factor* calculated as follows $P = 1 - e^{-2\tau}$ where τ is the optical depth. The effects of the adding the powder effect can be observed in Figures 4.14 and 4.15.

The only thing that changes throughout the cloud layer is the cloud density. We use two 3D Worley noise textures to simulate the changes in density throughout the medium. During raymarching, we sample these textures and multiply their value by the absorption coefficient. First we sample the higher resolution texture. This texture is used to generate the base low detail shape of the cloud. After this we sample the smaller texture and subtract it's value from the base texture producing detailed shapes of the cloud. Before subtracting, we weigh the detailed density by the inverse of the base density. This results in the detail being subtracted near the edges of the cloud, which is what we want.

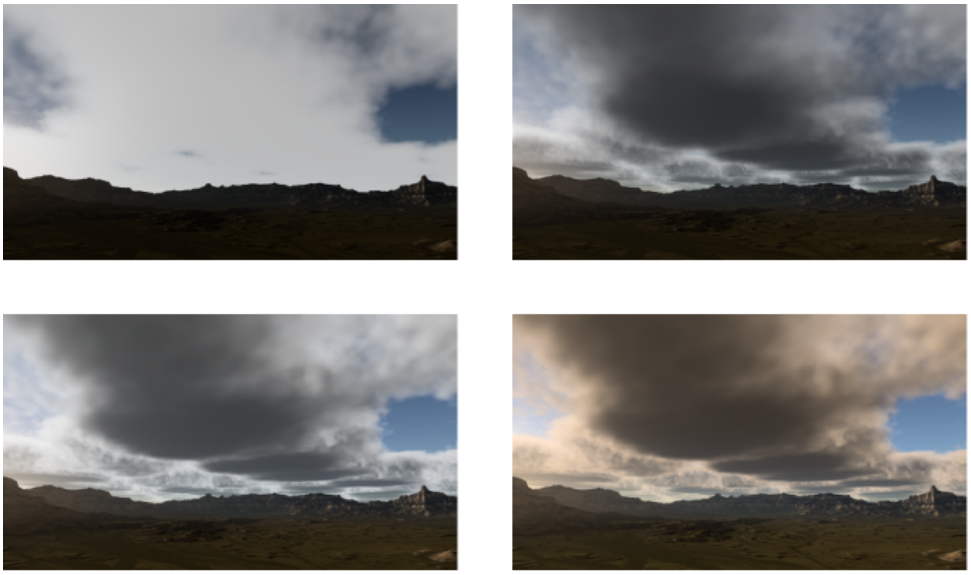


Figure 4.14: The effect of different components when ray marching clouds. Top left: no secondary ray march toward sun Top right: with secondary ray march towards sun Bottom left: secondary ray march to sun and powder effect Bottom Right: secondary ray march, powder effect and atmosphere transmittance

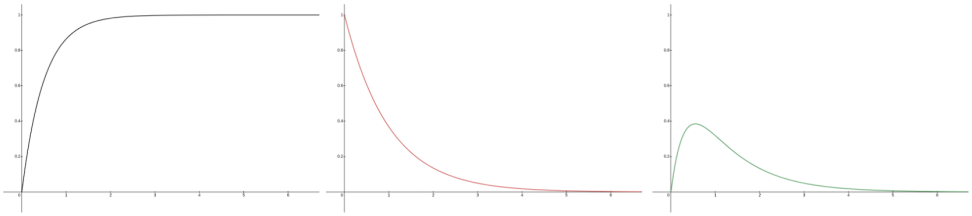


Figure 4.15: From left to right: (1) Classic transmittance calculation $T = e^{-\tau}$ (2) Powder effect calculation $P = e^{-2\tau}$ (3) (1) and (2) combined.

As already mentioned, in order to later correctly apply the aerial perspective effects on top of our clouds, we need to store write the cloud depth into attached depth buffer. We tried three different methods to calculate the depth of the cloud. Our first naive attempt was to simply store the depth of the first sample, which had transmittance less than a certain threshold. However, we found that this produced visible bands around the clouds. The resulting depth texture was also very blocky and did not capture the cloud details well. We then tried the solution proposed by [Hil16]. We store the depth of each sample weighed by the transmittance towards the view point. This is expressed by the following equation:

$$C_{depth} = \frac{\sum_{s=0}^{S} T(x_0, s) * Depth(s)}{\sum_{0}^{S} T(x_0, s)} \tag{4.3}$$

Where S is the amount of our samples, s is the sampled position, x_0 is the origin of the view point, and $(Depth(s))$ is the depth of the respective sample. This resulted in a much more detailed depth texture, but did not solve

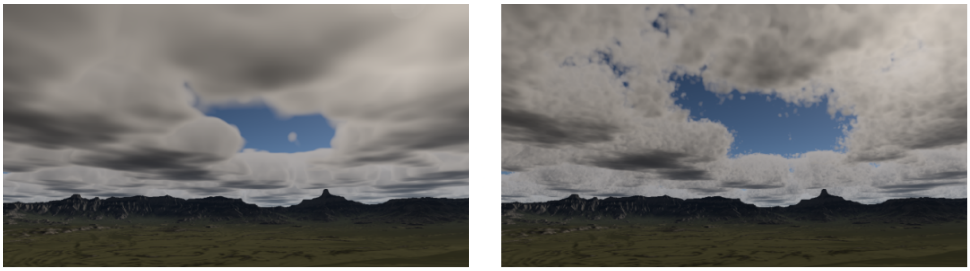


Figure 4.16: Left: Only the base shape sampled using the high resolution texture Right: Detail texture subtracted from the base shape

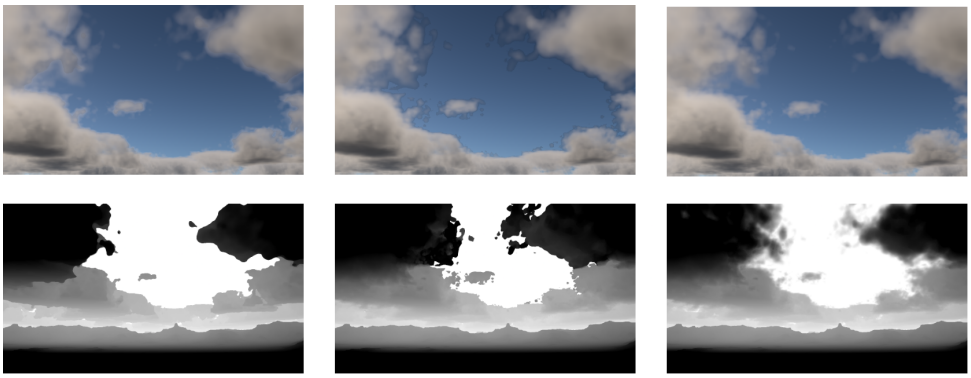


Figure 4.17: From left to right (1) Naive approach storing depth after some transmittance threshold (2) Storing depth as sample depth weighed by sample transmittance (3) Storing weighed depth and blending edges of clouds

the visible outlines. To remove the visible outlines, we linearly interpolate the maximum depth with the cloud depth based on the square of the final transmittance from the view position towards the final sampled position during our ray march. By doing this, we effectively blend the edges of the cloud with the far sky, producing smooth edges. The three depth textures and the resulting clouds can be seen in Figure 4.17

The final cloud color is then stored in the RGB channels. The alpha channel contains the final transmittance of the cloud. The clouds are then blended with the HDR backbuffer using the following equation:

$$FinalColor = BackgroundColor * CloudColor.A + CloudColor.RGB \tag{4.4}$$

Aerial Perspective

As a last step of the rendering process aerial perspective effects are applied to the entire scene. Since the Aerial perspective LUT is mapped to the camera frustum, we can use screen space coordinates which are the same as the Aerial perspective LUT coordinates. The only thing left to figure out is which layer we need to sample. We again use the method described previously in 4.5. By changing the NDC z value to the depth sampled from depth map written by Clouds pass, we get the real-world depth for the processed fragment.

This is then used to sample the Aerial perspective LUT and output it's value. Please note that we used the alpha channel to store the average transmittance. The blending equation used by this stage is given by the following equation:

$$FinalColor = BackgroundColor * (1.0 - AELUT.A) + AELUT.RGB \tag{4.5}$$

4.4.5 Post process Histogram and average luminance

The last stage before we present the image to the screen is the Post processing pass. As described above, we use two compute shaders to generate the average luminance, which is then used during tonemapping. We use the solution proposed by [Tar19].

The histogram is represented as a global array with the number of elements equal to the number of histogram values we want to store. We used the same size as proposed in the reference, dividing the luminance range into 256 segments. We utilized a local team of threads in a 16x16 block. Each thread reads a single value stored in the HDR image corresponding to the thread's global dispatch ID and converts it into luminance. The binary logarithm of this luminance value is then used to get the index of the corresponding segment in the histogram. We use a group local histogram copy to first accumulate all the values. Finally, we merge all local histograms into one single global histogram.

Listing 4.9: Code used to generate luminance histogram

```
1
2  shared uint HistogramLocal[256];
3  void main()
4  {
5      vec3 hdrColor = texture(HDRBackbuffer, uv).rgb;
6      uint binIndex = HDRToHistogramBin(hdrCol);
7      atomicAdd(HistogramLocal[binIndex], 1);
8
9      // wait for all threads in the group to finish
10     // writing into local histogram
11     groupMemoryBarrier();
12     barrier();
13
14     atomicAdd(HistogramGlobal[gl_LocalInvocationIndex],
15               HistogramLocal[gl_LocalInvocationIndex]);
16 }
```

During the second pass, we calculate the weighed sum of all the histogram entries, which is then converted into the average luminance for the current image. To calculate the sum, we use the same thread based merge which was used when merging Multiscattering LUT values 4.3. By subtracting this

Figure 4.18: In the top row we can the adaption effect when the scene suddenly becomes darker. In the bottom the opposite is happening the scene suddenly became overly brighter.

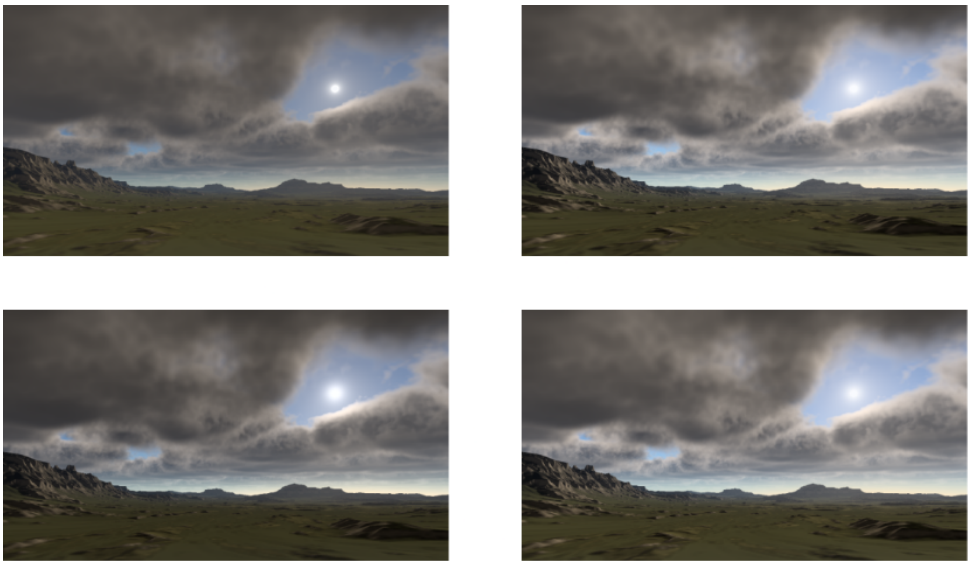


Figure 4.19: The effects of different tonemapping curves. Top Left: Reinhard Top Right: ACES Bottom Left: Lottes Bottom Right: Uchimura

new average luminance from the previously known value, we get the average luminance change between two frames. We then use the luminance change to slightly offset our old average luminance value. By only offsetting the old value instead of overwriting it, we achieve two effects. First, we avoid any artifacts when quickly changing scene luminance. Second, this approximates the effect of the eye slowly adapting to increasing (or decreasing) light intensity. This effect can be observed in Figure 4.18.

Lastly, the tonemapping shader is executed. The color value stored in the HDR backbuffer is converted from RGB space to xyY. Then it is remapped using the average luminance value computed previously and transformed using the tonemapping curve. We decided to expose four tonemapping curves 1. Reinhard tonemapping curve, 2. ACES Filmic Tone mapping curve [Nar16] 3. Lottes tonemapping curve [Lot16] and 4. Uchimmura tonemapping curve [US18] used in Gran Turismo. The comparison of these curves can be seen in Figure 4.19.

Chapter 5

Results

In this chapter, we present the results we obtained using our implementation. We also compare the performance of each scene on two different computers with the parameters provided in Table 5.1. Most of our frame budget was spent on raymarching clouds. Performance is highly dependent on the resulting cloud quality we want to achieve. In all of our benchmarks, we only had a single frame in flight. This slightly reduces the stability of the frame rate, but in return gives more consistent measurements.

Name	Type	CPU	GPU
PC1	Desktop	AMD RYZEN 7 1700	GeForce GTX 1080
PC2	Laptop	Intel Core i7 7700HQ	GeForce GTX 1050 (mobile)

Table 5.1: Computers and their hardware components performance was tested on.

5.1 Earth with no cloud cover

In the first scene, we would like to compare the look of the atmosphere without any cloud cover with real photos to see how accurate our model is. The scattering and extinction coefficients of the atmosphere used for this purpose can be seen in Table 5.2. We would also like to compare the aerial perspective effects of the atmosphere onto opaque objects. Multiple pairs of images will follow, each containing our rendered image and a real photo.

In Figure 5.1 we can see the comparison of sunset images. We can see that the overall palette is the same; however, slight changes are still present. The most striking difference is the lighting of the hills. Because we sample directly Transmittance LUT towards the Sun at each terrain fragment when the sun is behind the horizon, we get completely black terrain, which is not correct.

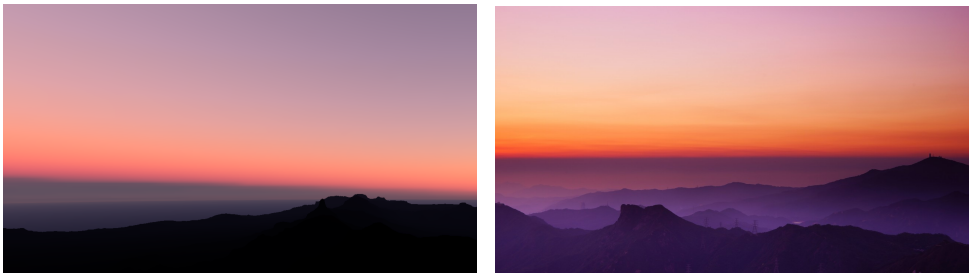
Figure 5.1 shows another comparison between the photo and the image rendered by our application. We can again see that ours has a similar color composition, but it is a bit darker. This could be caused by the manual

settings of the camera used to shoot the real photo versus our automatic adaptive luminance balancing.

Last Figure 5.3 shows comparison between photo and rendered image of clear day. Similarly to the previous comparison, our application produces a sky that looks darker. Along with this, the aerial perspective effects of the atmosphere can be clearly visible making the mountains look bluish white.

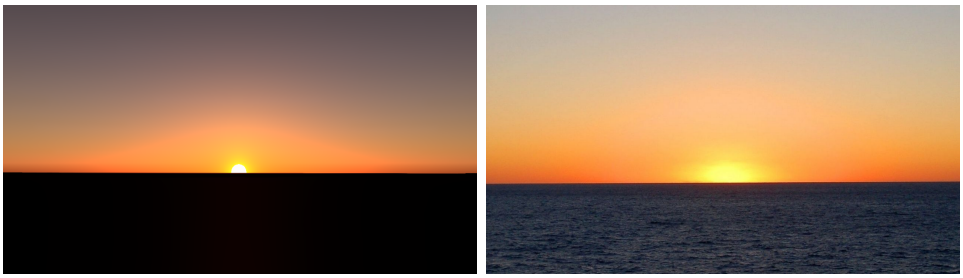
Coefficient	Value (km^{-1})	Distribution
β_R^{scat}	$(5.802 \cdot 10^{-3}, 13.558 \cdot 10^{-3}, 33.100 \cdot 10^{-3})$	$e^{\frac{-h}{8.0}}$
β_M^{scat}	$(3.996 \cdot 10^{-3}, 3.996 \cdot 10^{-3}, 3.996 \cdot 10^{-3})$	$e^{\frac{-h}{1.2}}$
β_M^{ext}	$(4.440 \cdot 10^{-3}, 4.440 \cdot 10^{-3}, 4.440 \cdot 10^{-3})$	$e^{\frac{-h}{1.2}}$
β_O^{ext}	$(0.650 \cdot 10^{-3}, 1.881 \cdot 10^{-3}, 0.085 \cdot 10^{-3})$	(h < 25 km) $\frac{h}{15} - \frac{2}{3}$ (h > 25 km) $\frac{8}{3} - \frac{h}{15}$

Table 5.2: Coefficients used to simulate atmosphere.



(a) : Image rendered by our application. **(b) :** Real photo of sunset (source Here)

Figure 5.1: Comparison between image rendered by our application and real photo of sunset.



(a) : Image rendered by our application. **(b) :** Real photo of sunset (source Here)

Figure 5.2: Comparison between image rendered by our application and real photo of sunset.

(a) : Rendered image of clear day. **(b) :** Photo of clear day (source Here).

Figure 5.3: Comparison between image rendered by our application and real photo of clear day.

5.2 Earth with medium cloud cover

Second test scene again uses earth like atmosphere setup. The atmosphere scattering and absorption coefficients are the same as in the previous scene and can again be seen in Table 5.2. We use a layer of clouds starting at 5.7 kilometers and ending at 8 kilometers. This is a bit higher than one would possibly find Cumulus clouds such as the ones we are trying to depict. We chose to place the clouds higher to compensate for the scale at which we are trying to display the scene. We can also see the effect of Aerial perspective in the second and third images when looking at the distant hills. The resulting images can be seen in Figure5.4 and the performance of individual shaders in Table 5.3. We can see that we are only barely hitting 60 fps on PC1, while in order to at least get close to 30 fps on PC2, we need to reduce the resolution to 720p.

Another thing to notice is that the main bottleneck is cloud rendering. It is usually responsible for over 60% of the total frame time. The LUT's are not screen resolution dependent; thus the time spent pre-calculating is almost the same for both 1080p and 720p. Additionally, the increase in time spent drawing atmospheric effects (not clouds) is also as decoupled as possible from the screen resolution. The increased cost when rendering Far Sky and Aerial perspective is only due to the increased amount of pixels we need to process and draw the sky texture on. The same holds for the Histogram computation as well as Tonemaping, because they also iterate over the whole screen.

	PC1		PC2	
Shader	1080p	720p	1080p	720p
Transmittance LUT	63.9 μs	63.9 μs	233.7 μs	235.4 μs
Multiscattering LUT	51.0 μs	51.1 μs	208.4 μs	210.0 μs
Sky View LUT	33.3 μs	32.4 μs	129.7 μs	130.6 μs
AE Perspective LUT	56.2 μs	56.4 μs	184.9 μs	186.5 μs
Draw Terrain	2.9 ms	2.9 ms	10.86 ms	11.73 ms
Draw Far Sky	216.7 μs	104.7 μs	979.9 μs	384.6 μs
Draw Clouds	11.6 ms	7.15 ms	43.7 ms	22.85 ms
Draw AE Perspective	278.3 μs	125.9 μs	1.26 ms	569.7 μs
Construct Histogram	228.1 μs	103.6 μs	415.3 μs	187.9 μs
Sum Histogram	3.3 μs	3.3 μs	3.9 μs	3.7 μs
Tonemap	341.0 μs	147.3 μs	1.12 ms	500.0 μs
Total	15.79 ms	10.73 ms	59.1 ms	36.99 ms

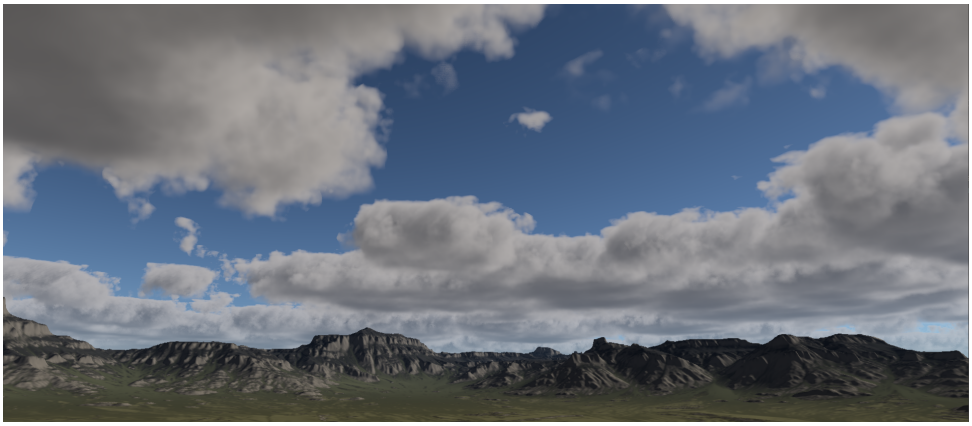
Table 5.3: Average execution times of each shader for earth like planet with clouds.

5.3 Fictional Planet

Last scene uses cloud and atmosphere parameters that are not based in reality. These settings are used to demonstrate the flexibility of our implementation. Given the procedural nature of our clouds combined with the parameterizable atmosphere, we are able to completely change the overall look and mood of the entire scene by tweaking a few values. Additionally, these fictional settings demonstrate the interconnected effects of both atmosphere and clouds producing consistent results even when we change the values outside of the ranges we are able to observe in the real world. The atmospheric parameters used to render this scene can be seen in Table 5.4.

We have raised the Mie scattering and extinction coefficients, as well as the Rayleigh scattering coefficient, by almost two orders. Together with the increase in the distribution of particles throughout the medium, we are able to simulate a very dense atmosphere. We achieved the purplish blue look of the atmosphere by leaving the blue-wavelength component of the Rayleigh scattering coefficient lower. As a result, most of the light in the red-green wavelength gets scattered away by the atmosphere before reaching the eye of the observer. To match the aerial perspective effects with the sky look, we also lowered the blue-wavelength component of the Mie absorption coefficient, allowing more blue light to penetrate the atmosphere.

Similarly to the previous scenes, the performance can be observed in Table 5.5. Most of the results stay the same with the one big exception of clouds. By raising the density of the clouds in the fictional scene, we are able to greatly reduce the number of raymarching samples used for each cloud.



(a) : Medium cloud cover with sun almost directly overhead.

(b) : Sparse cloud cover with sun nearing sunset. Aerial perspective effects are clearly visible on the terrain towards the sun.

(c) : Clouds during sunset. The clouds become lot darker as not a lot of sunlight reaches them through the atmosphere.

Figure 5.4: Images of clouds and atmosphere obtained by using Earth-like conditions.

This is due to the cloud shader terminating the raymarch earlier when a certain accumulated density threshold was reached. We are able to do this

Coefficient	Value (km^{-1})	Distribution
β_R^{scat}	(0.149, 0.142, 0.051)	$e^{\frac{-h}{18.416}}$
β_M^{scat}	(0.046, 0.047, 0.057)	$e^{\frac{-h}{13.143}}$
β_M^{ext}	(0.082 , 0.071, 0.058)	$e^{\frac{-h}{13.143}}$
β_O^{ext}	$(0.650 \cdot 10^{-3}, 1.881 \cdot 10^{-3}, 0.085 \cdot 10^{-3})$	(h < 25 km) $\frac{h}{15} - \frac{2}{3}$ (h > 25 km) $\frac{8}{3} - \frac{h}{15}$

Table 5.4: Coefficients used to simulate the atmosphere on a fictional planet.

because for very high accumulated densities the amount of light increase with each additional sample is negligible and thus has almost no visual impact on the final result.

These optimizations can also introduce artifacts, resulting in visually unappealing results whenever we increase the cloud density too much. In the case of this scene it was possible to raise the cloud density quite high because the dense atmosphere hides away most of the imperfections produced.

	PC1		PC2	
Shader	1080p	720p	1080p	720p
Transmittance LUT	64.2 μs	63.9 μs	234.8 μs	236.6 μs
Multiscattering LUT	51.1 μs	51.2 μs	209.3 μs	211.0 μs
Sky View LUT	32.7 μs	32.3 μs	130.2 μs	131.2 μs
AE Perspective LUT	56.8 μs	56.4 μs	185.8 μs	187.2 μs
Draw Terrain	2.86 ms	2.88 ms	10.31 ms	10.44 ms
Draw Far Sky	216.2 μs	101.6 μs	896.1 μs	441.6 μs
Draw Clouds	7.89 ms	5.13 ms	22.52 ms	13.75 ms
Draw AE Perspective	277.76 μs	125.4 μs	1.26 ms	569.7 μs
Construct Histogram	227.2 μs	103.0 μs	418.0 μs	186.1 μs
Sum Histogram	3.3 μs	3.3 μs	3.8 μs	3.7 μs
Tonemap	342.3 μs	146.4 μs	1.13 ms	494.0 μs
Total	12.02 ms	8.69 ms	37.3 ms	26.96 ms

Table 5.5: Average execution times of each shader for the fictional planet.

(a) : Dense cloud cover with big cloud layer.

(b) : Sun is near horizon wihtout any clouds. The blue tinted atmosphere is very clearly visible.

(c) : Sun is near horizon and clouds are lower than in Figure 5.5a. Most of the light is absorbed by the atmosphere before it reaches the cloud layer.

Figure 5.5: Images of clouds and atmosphere obtained by using Earth-like conditions.

Chapter 6
Conclusion

In this work, we have described the implementation of several techniques combined to create a single interconnected system to render atmospheric effects. By using Vulkan API we were successfully able to leverage GPU for most of our computations and thus reach real-time frame rates. The model previously described by Hillaire [Hil20] was used to render the sky. On top of this, a technique presented by Shneider [SV15] was implemented, allowing us to combine the atmosphere model with procedurally generated clouds.

The implemented solution allows visualization of miscellaneous settings ranging from those based in reality all the way to those completely fictional. Although this implementation relies on the use of multiple LUTs, it is still possible to change almost all the parameters during the run-time of the application.

6.1 Future work

The most pressing issue of the implementation presented is the performance of rendering clouds. First, due to the fact that clouds are raymarched for each pixel of the resulting image, there is a dependency on the resolution of the final image. One possible improvement would be to render the clouds into a fixed-sized texture (similarly to how the sky view LUT texture is generated), which would then be sampled and up-scaled to fit the final resolution. Additionally, we could reduce the amount of raymarching needed for each frame by updating only one out of sixteen pixels for each 4x4 block of pixels for each frame. Combining this by reprojecting the previous frame would keep the visual quality high while significantly reducing the amount of raymarching done each frame. Secondary benefit of using this approach would be the option to use noise for offsetting start of each ray at the start of raymarching more aggressively, which would allow us to use fewer samples and reduce the raymarch complexity even further.

Another thing missing from our application are shadows. We believe that by adding hard as well as soft volumetric shadows one could improve the appeal and realism of the resulting images significantly.

Bibliography

[BN08] Eric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. *Computer Graphics Forum*, 27(4):1079–1086, 2008.

[BNM+08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 173–182, 2008.

[CBE+21] Jonathas Costa, Alexander Bock, Carter Emmart, Charles Hansen, Anders Ynnerman, and Claudio Silva. Interactive visualization of atmospheric effects for celestial bodies. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):785—-795, 2021.

[Gon09] Álvaro González. Measurement of areas on a sphere using fibonacci and latitude–longitude lattices. *Mathematical Geosciences*, 42(1):49–64, 2009.

[Hil16] Sbastien Hillaire. Physically based sky, atmosphere and cloud rendering in frostbite. In *ACM SIGGRAPH*, 2016.

[Hil20] Sébastien Hillaire. A scalable and production ready sky and atmosphere rendering technique. *Computer Graphics Forum*, 39(4):13—22, 2020.

[HKS98] Michael Hess, Peter Koepke, and Ingrid Schult. Optical properties of aerosols and clouds: The software package opac. *Bulletin of the American meteorological society*, 79(5):831–844, 1998.

[HW12] Lukas Hosek and Alexander Wilkie. An analytic model for full spectral sky-dome radiance. *ACM Transactions on Graphics (TOG)*, 31(4):1–9, 2012.

[KMM+17] Simon Kallweit, Thomas Müller, Brian Mcwilliams, Markus Gross, and Jan Novák. Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks. *ACM Transactions on Graphics (TOG)*, 36(6):1–11, 2017.

[Kok04] Alexander Kokhanovsky. Optical properties of terrestrial clouds. *Earth-Science Reviews*, 64(3-4):189–241, 2004.

[Lag19] Sebastian Lague. Coding adventure: Clouds. https://www.youtube.com/watch?v=4QOcCGI6xOU, 2019.

[Lot16] Timothy Lottes. Advanced techniques and optimization of hdr color pipelines. Game Developers Conference, 2016.

[Nar16] Krzysztof Narkowicz. Aces filmic tone mapping curve. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/, 2016.

[Ove21] Alexander Overvoorde. Frames in flight, vulkan tutorial, 2021.

[Pet06] Grant W. Petty. *A first course in atmospheric radiation*. Sundog Pub., 2006.

[SV15] Andrew Schneider and Nathan Vos. The real-time volumetric cloudscapes of horizon: Zero dawn. *SIGGRAPH Course note in Advance in Real-Time Rendering in Games*, 2015.

[Tar19] Alex Tardif. Adaptive exposure from luminance histograms. https://www.alextardif.com/HistogramLuminance.html, 2019.

[US18] Hajime Uchimura and Kentaro Suzuki. Practical hdr and wide color techniques in gran turismo sport. In *SIGGRAPH Asia 2018 Courses*, pages 1–4. 2018.

[WVBR+21] Alexander Wilkie, Petr Vevoda, Thomas Bashford-Rogers, Lukáš Hošek, Tomáš Iser, Monika Kolářová, Tobias Rittig, and Jaroslav Křivánek. A fitted radiance and attenuation model for realistic atmospheres. *ACM Transactions on Graphics (TOG)*, 40(4):1–14, 2021.

[Yus14a] Egor Yusov. High performance outdoor light scattering using epipolar sampling. *GPU Pro*, 5:101–126, 2014.

[Yus14b] Egor Yusov. High-performance rendering of realistic cumulus clouds using pre-computed lighting. In *High Performance Graphics*, pages 127–136, 2014.

Appendix A

Building and controlling the application

A.0.1 Building the application

Our application uses Premake to create a project or makefile which is then used to generate the executable.

When building on Linux, the following dynamic libraries have to be installed: glfw and vulkan-dll

The steps to successfully use the application on Linux are as follows.

1. Download and install Vulkan SDK and GLFW.
2. Go to the root directory and from the command line call **premake5 gmake**.
3. From the command line call **compile_shaders_linux.sh**.
4. From the command line call **make config=debug_linux**.
5. Run the executable stored in bin/Linux/Debug/Atmosphere

When building on Windows, almost all the required resources are included. The only thing missing is the Vulkan SDK along with the glslc compiler used to compile shaders into SPIRV. Please note that not all features are fully working on Windows as of the time of writing this document. These problems are likely to be fixed in the future. You can find the most recent version of this application on my GitHub.

The steps to successfully use the application on Windows are as follows.

1. Download and install the Vulkan SDK from Here.
2. Go to the root directory and from the command line call **premake5 vs2022** (or the version of Visual Studio that you are using).
3. From the command line call **compile_shaders_windows.bat**.
4. Open the generated visual studio solution.
5. Switch the solution platform from Linux to Windows.

A.0.2 Controls

The application operates in two modes. The first is the control mode. This is the mode in which the application is initially in. This mode allows the use of WASD keys to move in the world along with the cursor enabled. This mode is meant to be used to interact with the GUI and change the parameters of the atmosphere and clouds. By pressing the key **f** on the keyboard, the mode is switched to fly-through mode. The control scheme in this mode is slightly different:

- WASD move forward, left, back and right, respectively.
- SPACE increase altitude.
- CTRL decrease altitude.

In addition to this, the cursor is disabled in the fly-through mode, and the user can control the view direction with the usage of mouse. To switch back to the control mode, the f-key has to be pressed again.

Appendix B

Attachment Structure

In this Appendix, we describe the structure of the attachment application source code.

- **bin** - this folder is used to store the generated binary files when using make to build the application
- **dependencies** - contain the libraries required to successfully build the application on Windows.
- **obj** - contains the intermediate object and dependency files used when building the application using make
- **shaders** - contains all shader-related files.
  - **build** - mirrors the shade source code structure and contains all of the intermediate SPIRV files
- **source** - contains the entire source code of our application.
  - **model** - Source files used to fill and control Atmosphere values for sending to the GPU
  - **noise** - Source files used to generate 3D worley noise on the GPU
  - **vendor** - Source files for libraries which were used in our application
  - **vulkan** - Contains source files of the Vulkan abstraction and renderer
- **assets** - Folder containing all assets used in our application