Real-time Rendering of Atmosphere and Clouds in Vulkan

Paper 9

Abstract

This work presents a Vulkan-based implementation rendering volumetric clouds and atmosphere. We combine previously published solutions to produce a single unified look. We use Raymarching as the main method to render both the atmosphere and clouds. Furthermore, we use multiple precomputed look-up tables (LUTs) proposed by Hillaire to speed up the rendering of the atmosphere. We enhance these methods with the option to render volumetric clouds using a precomputed three-dimensional texture setup storing procedurally generated noise. With our final solution, we can render images in a high dynamic range. We apply post-processing effects and use adaptive luminance to transform the image into a low dynamic range for presentation.

Keywords: Volumetric clouds, Real-time rendering, Atmosphere, Vulkan

1 Introduction

Having a realistic and believable atmospheric model when rendering dynamic environments in interactive applications such as games 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. In addition, these effects are also interconnected and affect each other, making them even harder to simulate. Despite the gradual increase of computing power available in personal computers, simulating complex light interactions that produce the visual appearance of sky and clouds is still very difficult. Paired with the constraint of displaying such effects in real time, this problem becomes even harder. To avoid these problems, we are forced to adapt number of approximations, gaining a big reduction in the problem complexity.

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.

In Section 2, we describe various approaches to rendering volumetric media along with their strengths and weaknesses. A short summary of the relevant topics in physics follows in Section 3 [1]. Lastly in Sections 4 and 5 we propose the solution and describe our implementation.

[1] For brevity sake we only provide a short description meant as a reference. More detailed descriptions along with explanations can be found, for example, in [10].

2 Background

As we focus on real-time rendering, we will only describe methods that are relevant in this context. The most physically accurate method to render volumetric effects such as sky and clouds is to use path tracing [7] [9]. 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 given by the number of rays we need to trace to reduce the noise in the final image.

Another approach using fitted mathematical models has also been proposed [6] [14]. Methods leveraging this principle usually use measured data to build a set of parameters used to evaluate the sky look. These models are very fast; however, due to the dependence on the data measured in the real world, these methods do not provide the option to change the parameters of the atmosphere. In addition, when the parameters of the atmosphere are changed, a new model has to be fitted. As the goal of this work is rendering parameterizable atmospheres reflecting the conditions present on other planets as well as on Earth, these methods did not fit our requirements.

Finally, many methods use raymarching to achieve their results [2]. 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. For these reasons, we chose to use ray marching in our implementation as well.

2.1 Atmosphere

The first methods for rendering physically based atmospheres focused only on single scattering evaluation by ray marching the atmosphere from viewpoint for each pixel on the screen [12]. Although evaluating only single scattered light has performance benefits, it struggles to represent realistic looking atmospheres. This is especially problematic when trying to represent more dense atmospheres, which results in dark and unrealistic scenes. Due to this, methods that take into account multiple scattering were introduced [3] [15].

Such methods usually 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

the evaluation. This significantly improves the rendering time. Where previously the same evaluation was repeated hundreds of times, now it is only computed once at the beginning. The results are then accessed whenever needed. 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. Another disadvantage is that, because the results are obtained by raymarching each pixel, the performance is tied to the resolution of the screen.

Hillaire [5] introduced a solution to overcome the above-mentioned problems. 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 the 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. This effectively decouples the computation complexity from window resolution and introduces additional speed improvements. Bruneton [2] provides a good summary and comparison of different sky models.

2.2 Clouds

We summarize previous approaches to rendering clouds that are most interesting or relevant to our work. One possible approach was to represent clouds as volumes of particles. For example, Yusov [16] presented a particle-based rendering method. The clouds were modeled using randomly rotated and scaled copies of a single reference particle. The complex optical properties of the reference particle were precomputed, making this process viable for use in real-time applications.

Another technique was presented by Bouthors et al. [1]. By combining meshes to represent low resolution cloud boundaries together with procedural volumetric hypertextures, which add the detail under the mesh boundary, an efficient cloud representation was achievable. When rendering, the cloud surface is covered with circular collectors that are used to evaluate the incoming light. Using this information along with a set of pre-computed transfer tables, the light is integrated. The cloud representation, however, is not trivial to tweak. This, along with the relatively high overall complexity of the described method, is why simpler methods were developed.

The more recent work by Schneider [11] 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.

3 Physical model

Light transport in participating media is a well-studied problem in computer graphics, described in detail in many articles. This section summarizes the fundamentals of light propagation in the atmosphere relevant to this work and is strongly motivated by works [4][5][10].

When electromagnetic radiation travels through the atmosphere, it collides with the molecules that make up the atmosphere. During this collision, part of the energy carried by the radiation is absorbed, part is reflected (scattered), and part is emitted. 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

$$\beta_a = \frac{4\pi n_i}{\lambda} \tag{1}$$

where λ 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.

$$\beta_e = \beta_a + \beta_s \tag{2}$$

To correctly compute attenuation over a path where the extinction coefficient varies, integrating the coefficient along the entire path of the ray is required. So, 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 equation 3

$$L(\lambda, x_2) = L(\lambda, x_1) \exp\left[-\int_{x_1}^{x_2} \beta_e(x)\, dx \right] \tag{3}$$

where λ is the wavelength of the radiation considered.

$$T(x_1, x_2) = e^{-\int_{x_1}^{x_2} \beta_e(x)\, dx} \tag{4}$$

We also have to consider the scattering effects of the atmosphere. Unlike absorption effects, when radiation is scattered away, it is added to the atmosphere at a different point. The radiation scattered away is not uniform in all possible directions. To represent the directional distribution of the scattered light, we use a *scattering phase function* denoted by $P(cos\theta)$ where θ is

$$\cos\theta = \overrightarrow{\omega}' \cdot \overrightarrow{\omega} \tag{5}$$

with $\overrightarrow{\omega}'$ being the incoming direction of the light and $\overrightarrow{\omega}$ being the direction of the ray we consider. We can make the scattering phase function depend only on the parameter Θ because particles in the atmosphere are spherical or randomly oriented.

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

$$dL_{scat}(\lambda, x, \overrightarrow{\omega}) = \int_{4\pi} \beta_s(y) P(cos\Theta) L(\lambda, x, \overrightarrow{\omega}') d\overrightarrow{\omega}' ds \tag{6}$$

By combining the two previously described effects (Eq. 3 and Eq. 6), we get the following form.

$$L(\lambda, x, \overrightarrow{\omega}) = \underbrace{T(x, x_0) L(\lambda, x_0, -\overrightarrow{\omega})}_{direct\ light\ \ from\ sun} + \underbrace{\int_x^{x_0} \beta_s(y) T(x, y) \int_{4\pi} P(cos\Theta) L(\lambda, y, \overrightarrow{\omega}') d\overrightarrow{\omega}' dy}_{in-scattered\ light\ along\ the\ ray} \tag{7}$$

Where λ is the wavelength of the radiation considered, x is the origin of the ray, and $\overrightarrow{\omega}$ is the direction of the ray. The part marked *direct light from sun* almost directly corresponds to Equation 3. We rewrote the second part, corresponding to the in-scattered light, as follows. First, we can take the coefficient β_s from the inner integral, as it remains constant over the integrated area. Second, since we consider in-scattered light along a ray [2], we integrate over the entire ray and weigh the results by transmittance.

Next, we will describe two models used to substitute the real scattering phase function $P(cos\Theta)$. First, for particles that are much smaller than the wavelength of incident radiation, such as clear air molecules or ozone, the Rayleigh scattering phase function is used. We use the model proposed by [4]

$$P_R(\theta) = 0.7629(1 + 0.932 \cdot cos^2(\theta)) \cdot \frac{1}{4\pi} \tag{8}$$

Second, for particles that are comparable to or larger than the wavelength of the incident radiation [3] Mie's theory was used. Larger particles, such as aerosols, tend to scatter light strongly forward. We use the double Henyey-Greenstein phase function approximation proposed again by [4]

$$P_M(\theta, g_1(\lambda), g_2(\lambda), \alpha(\lambda)) = \alpha * P_{fb}(...) + (1 - \alpha) * P_{fb}(...) \tag{9}$$

$$P_{fb}(\theta) = \frac{(1 + g_1^2(\lambda))}{(1 + g_1^2(\lambda) - 2g_1(\lambda) cos(\theta))^{\frac{3}{2}}} \tag{10}$$

For more details on scattering or extinction coefficients, see [4].

[2] as opposed to Equation 6 where we consider in-scattered light at a single point

[3] An example of such particles in atmosphere can be dust or water droplets

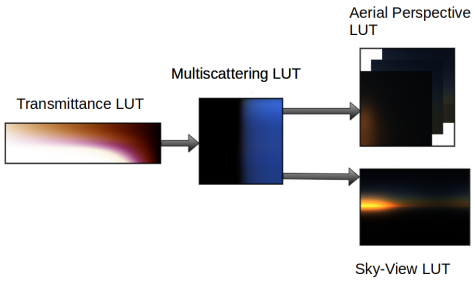

Figure 1: The order in which individual Sky LUTs are drawn. Please note that color values of LUTs have been scaled in order to be properly visible.

4 Proposed solution

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. We use the LUT setup proposed by [5], four LUTs storing precomputed parts of Equation 7. Individual LUTs and their dependencies can be seen in Figure 1.

4.1 Atmosphere precomputations

Transmittance LUT introduced by [3] is used to store the transmittance T described by Equation 4. When the atmosphere is ray-marched, the value of T is used very frequently to model the atmosphere attenuating the passing light. To compute this value, a second ray must be traced towards the light source. Given the overall smooth distribution of the atmospheric transmittance, it is beneficial to precompute the transmittance value for the entire atmosphere and store it in a table.

For **Multiscattering LUT** a new approach proposed by Hillaire [5] was used. We precompute the scattering contribution denoted by Equation 6 at several discrete points in the atmosphere. The incoming radiance from the Sun ($L(\lambda, x, \overrightarrow{\omega}$‘) in Equation 6) should be weighed by the transmittance. Here, we use the Transmittance LUT to retrieve the values instead of computing them directly.

Sky-View LUT represents the far sky mapped into a latitude/longitude texture that is much lower in resolution than the final image. This LUT stores the values of Equation 7. Similarly to Multiscattering LUT we use the Transmittance LUT to retrieve transmittance values instead of computing them. Additionally, we also interpolate values stored in Multiscattering LUT when marching each ray instead of computing the part of Equation 7 marked as ”in-scattered light along the ray”. The highest visual fre-

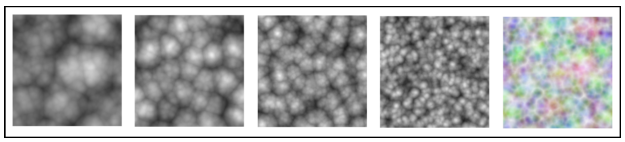

Figure 2: From left to right separate RGBA channels storing Worley noise and all of the channels combined together in the right most image.

quency is introduced by the Sun. Thus, we orient the Sky-View LUT so that the sun is always present at the same position in the texture. We map the values non-linearly, by adding more samples near the horizon.

Lastly, we use **Aerial (AE) perspective LUT** . The Aerial perspective refers to how we see objects as they recede into the distance from the viewpoint. 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 luminance of the atmosphere (Equation 7) and the average transmittance at the corresponding depth (Equation 4). Similarly to Sky-View LUT at each depth level we use the results stored in Transmittance and Multiscattering LUTs to speed up the computation.

4.2 Rendering process

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 part consists of computing the four LUTs used to render the atmosphere. The second part draws all of the scene objects and terrain. Along with this the atmosphere, it's effects and clouds are also rendered. The next step is to map the values from the HDR range into LDR that is used by the image presented to the screen. Finally, the step renders the user interface that controls various parameters of the atmosphere and clouds.

We also have a fifth standalone part, which is to compute the Worley noise texture later used to draw the clouds. We reuse this texture and do not recompute it each frame (which can be quite expensive).

The very popular method of rendering clouds by [11], 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 are randomly distributed in a desired volume for 3D texture.

After this, for each 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 voxel and $distance$ is the distance from the currently processed voxel towards the nearest point. We precompute multiple 3D textures that contain Worley noise with various frequencies. These textures are then sampled by raymarching the cloud.

Look up table	Resolution	size
Transmittance LUT	256×64	128 KiB
Multiscattering LUT	32×32	8 KiB
Sky-View LUT	192×128	198 KiB
Aerial Perspective LUT	$32 \times 32 \times 32$	256 KiB
Total		590 KiB

Table 1: Parameterization and sizes of LUT's used to render atmosphere.

We follow the method proposed by [8]. It uses two 4-channel 16 bit 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. These textures can be seen in Figure 2.

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.

5 Implementation

As in most performance-dependent applications, this work was implemented using C++. Vulkan API was used as an interface with the GPU.

5.1 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.

When rendering the atmosphere, four previously described LUTs have to be computed. We use a 16 bit RGBA texture for each LUT. The parameterization that we decided to use can be seen in Table 1.

In addition to the above, we use two volumetric textures, **Base Noise LUT** and **Detail Noise LUT**, which 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 2. The resolution of the noise textures is slightly higher than described by [11]. 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
Base Noise LUT	$256 \times 256 \times 256$	128 MiB
Detail Noise LUT	$128 \times 128 \times 128$	16 MiB
Total		144 MiB

Table 2: Parameterization and LUT sizes used to store Worley noise.

5.2 Main draw loop

Because most of the command buffers in our implementation are prerecorded and do not need to be reconstructed, the main purpose of the draw loop is to keep the GPU fed with as much work as possible. In order to do this, we have multiple *frames in flight*. By default, in our implementation, two frames are in flight at the same time.

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.

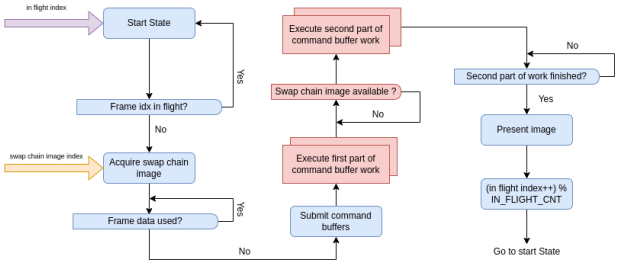

Figure 3: Flow diagram of the draw loop execution order. CPU parts as well as CPU-GPU synchronization points are colored blue. Similarly GPU parts and GPU-GPU synchronization are colored red.

Because swapchain images might be returned out of order, 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 swapchain, we check the corresponding frame data structure fence. Only after the fence has been signaled is an appropriate command buffer submitted to GPU. Whenever we finish rendering any frame, a structure fence is signaled, allowing another frame to be submitted. Figure 3 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. Since the second pair of command buffers writes directly into a swap chain image, we need to make sure that the corresponding image is available for us to write. We use an additional array of semaphores.

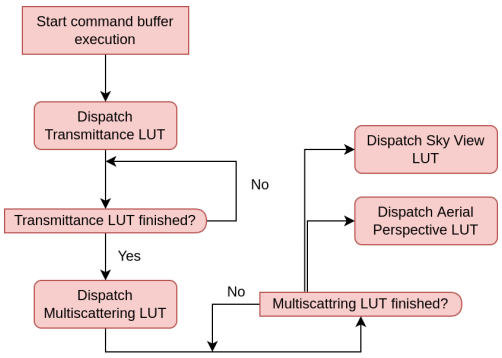

Figure 4: Flow diagram showing dependencies between individual operations in LUTs command buffer. As all parts are either executed on GPU or are GPU-GPU synchronization they are all marked red.

Each semaphore is signaled when the presentation engine is finished using the corresponding image.

5.3 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.

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 by one shader in one dispatch.

The compute dispatch commands are recorded in the order shown in Figure 1 in the command buffer. Because there are data dependencies between individual LUTs, we need to introduce synchronization between the individual dispatch commands. We use pipeline barriers after each dispatch, waiting after each drawcall. This is to ensure that all of the compute work previously submitted has been finished before issuing another dispatch. Figure 4 shows the visualization of the execution order in this command buffer, as well as the synchronization performed between executions.

5.3.1 Worley noise command buffer

As mentioned above, sometimes an additional LUT command buffer may be submitted that computes the 3D Worley noise textures. We again have no use for graphics pipeline's features when generating this texture and opt for compute shaders.

Following the approach used by [8] in the first pass, we

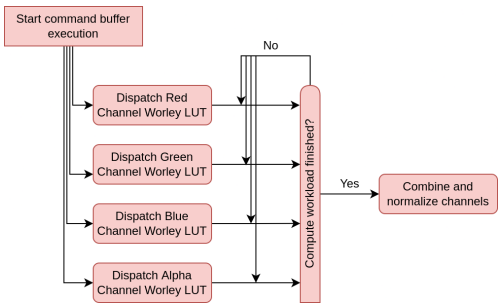

Figure 5: Flow diagram showing dependencies between and execution order of Worley noise command buffer.

render Worley noise, followed by the second pass, which normalizes the values in the range between 0 and 1. We used a separate one-channel texture for each of the final four textures in the first pass. These four textures are combined into a single 4-channel texture in the second pass.

The only synchronization that we have is between the first and second passes. In our case, a single pipeline barrier is inserted. This makes sure that all writes and reads in the 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 5.

5.3.2 Sky command buffer

Next we render all objects in our scene and draw the sky with clouds. The ordering here is important; we first render all objects before drawing the sky and clouds. This is because drawing the sky, clouds, and atmosphere requires depth information about the rest of the scene.

After all scene objects were rendered far sky is drawn where no object was drawn in previous pass. Then, the clouds are rendered. Lastly, the aerial perspective is applied. We used the depth when raymarching the clouds as well as the index into the aerial perspective LUT.

In order for aerial perspective to correctly apply on clouds, we also need information about how far the clouds are from the viewing point stored in the depth texture. Because Vulkan does not allow a read and write the same texture from the same shader. We introduce a second depth texture. When rendering clouds, use the first depth texture as an input and combine it with the depth of the rendered clouds. We write this result into the second output texture. Figure 6 shows how every pass reads or writes resources from the framebuffer.

We divide this command buffer into multiple subpasses and use subpass dependencies for image transitions as well as synchronization. Each subpass is responsible for one of the phases described above. After each subpass finishes a set of barriers corresponding to Figure 6 is executed.

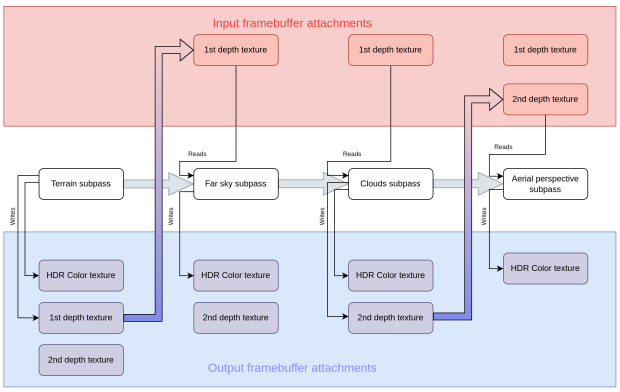

Figure 6: Visualization of writes and reads performed by each pass. Additionally we also show when do transitions from output attachment into input attachment inside a framebuffer occur.

5.3.3 Post process and GUI command buffers

Finally, at the end of each frame, the post-processing of the final image and the drawing of the UI follow. The UI allows the user to tweak almost all parameters used while rendering the sky. Position of the Sun in the sky, scattering and absorption coefficients, atmosphere height, and falloff of Rayleigh and Mie particle density. As for the clouds, the height where the could layer starts, as well as the thickness of the cloud layer, phase function parameters, and finally weights and scales of the noise textures used. It is also possible to tweak various tonemapping parameters relevant to each tonemapping curve.

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 [13].

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.

A pipeline barrier is inserted between the construction of the histogram and the calculation of the average luminance. A second pipeline barrier is inserted before tonemapping to ensure that the previous average luminance was written.

The last command buffer draws the UI on the screen. We do not need any synchronization, everything is handled internally by the ImGUI implementation.

6 Results

In this chapter, we present the results obtained by using our implementation. We also provide the performance for Earth-like setup. The scene was tested on two computers - PC1 with AMD RYZEN 7 1700 & NVIDIA GTX 1080 and PC2 with Intel Core i7 7700HQ & NVIDIA

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

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

Figure 7: Images of clouds and atmosphere obtained by using Earth-like conditions.

GTX 1050 (mobile). Most of our frame budget was spent on raymarching clouds (see Table 3). 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.

6.1 Earth with medium cloud cover

Our first testing scene was an Earth-like atmosphere setup. We can see that we are only barely hitting 60 frames-per-second on PC1. The execution times of individual shaders can be seen in Table 3. The main bottleneck is the cloud rendering, which is expected as we raymarch each pixel. The resulting images can be seen in Figure 7.

6.2 Fictional planet

Second scene uses cloud and atmosphere parameters that are not based on reality. These settings are used to demonstrate the flexibility of our implementation. Given the procedural nature of our clouds combined with the parameterizeable 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 rendered images of the fictional setup can be seen in Figure 8.

	PC1		PC2
Shader	1080p	720p	1080p
Transmittance LUT	63.9 μs	63.9 μs	233.7 μs
Multiscattering LUT	51.0 μs	51.1 μs	208.4 μs
Sky-View LUT	33.3 μs	32.4 μs	129.7 μs
AE Perspective LUT	56.2 μs	56.4 μs	184.9 μs
Draw Terrain	2.9 *ms*	2.9 *ms*	10.86 *ms*
Draw Far Sky	216.7 μs	104.7 μs	979.9 μs
Draw Clouds	11.6 *ms*	7.15 *ms*	43.7 *ms*
Draw AE Perspective	278.3 μs	125.9 μs	1.26 *ms*
Construct Histogram	228.1 μs	103.6 μs	415.3 μs
Sum Histogram	3.3 μs	3.3 μs	3.9 μs
Tonemapping	341.0 μs	147.3 μs	1.12 *ms*
Total	15.79 *ms*	10.73 *ms*	59.1 *ms*

Table 3: Average execution times of each shader for earth like planet with clouds.

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.

7 Conclusion and future work

We have described the implementation of several techniques combined to create a single interconnected system to render atmospheric effects. Using the Vulkan API, we successfully leveraged GPU for most of our computations, and thus reached real-time frame rates. The model described previously by Hillaire [5] was used to render the sky. In addition to this, a technique presented by Schneider [11] was implemented, which allows us to combine the atmosphere model with procedurally generated clouds.

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

The most pressing issue of the implementation presented is the performance of rendering clouds. The current cloud raymarching implementation is naive, and we

(a) Dense cloud cover with thick cloud layer.

(b) Sun is near horizon most of the light is absorbed by the atmosphere before it reaches the cloud layer.

Figure 8: Images of clouds and atmosphere obtained by using Earth-like conditions.

believe optimizing would be a promising direction. Alternatively, we expect that adding hard and soft volumetric shadows along with godrays could improve the appearance and realism of the resulting images significantly.

References

[1] 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.

[2] Eric Bruneton. A qualitative and quantitative evaluation of 8 clear sky models. *IEEE transactions on visualization and computer graphics*, 23(12):2641–2655, 2016.

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

[4] 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.

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

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

[7] Eric P Lafortune and Yves D Willems. Rendering participating media with bidirectional path tracing. In *Rendering Techniques' 96: Proceedings of the Eurographics Workshop in Porto, Portugal, June 17–19, 1996 7*, pages 91–100. Springer, 1996.

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

[9] Jan Novák, Andrew Selle, and Wojciech Jarosz. Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 33(6), November 2014.

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

[11] 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.

[12] Jaroslav Sloup. A survey of the modelling and rendering of the earth's atmosphere. In *Proceedings of the 18th spring conference on Computer graphics*, pages 141–150, 2002.

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

[14] 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.

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

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