School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Natural Phenomena for a First-Person Game Engine

Aleksandrs Zizkuns

Submitted in accordance with the requirements for the degree of MEng High Performance Graphics and Games Engineering

2019/2020

30 Credits

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Project report	*Report in a PDF document*	Prof Raymond Kwan (assessor) Dr Hamish Carr (supervisor) 05/06/2020
C++ OpenGL Demo Application	*URL to GitHub Repository https://github.com/koladonia/Natural-Phenomena*	Dr Hamish Carr (supervisor) 05/06/2020

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) 

Summary

Initially the project intention was to create a small game engine in which techniques that produce photorealistic effects could be easily integrated to be shown as a part of the demo, with the main focus being on techniques and, to a lesser extent, software engineering. As there was a number of unexpected issues external to the project like change of the number of contributors to the project, health condition of one of the contributors and global pandemic, there was a drastic change in its main focus with a much reduced scope to accommodate the situation.

This project aims to demonstrate modern techniques used to render atmospheric effects in real-time found in game engines like Frostbite and Decima. It explores options and physical model approximations to give a visually appealing and realistic image without sacrificing much performance so that the computation runtime of these effects would not be the performance bottleneck of the whole application.

One of the key parts of the project is to give the evaluation of the used algorithms and suggest points upon which further work could be built.

Acknowledgements

I am very grateful to Dr Hamish Carr for dedicating his time to resolve whatever issues that could arise during my time at the university and as an external student.

Also, I would like to thank my family, friends, medical staff, and every single person that supported me during the hardships of recent years.

Table of Contents

Chapter 1
Introduction

1.1 Goals

The project aim was to develop a First Person Game Engine complete with user interface, scripting capabilities and possible inclusion of various engine support systems for external assets and internal resource management, physics, terrain generation, lighting effects, mesh optimization (both realtime and preprocess), procedural level generation, AI and networking, and a small demo that fully utilizes its features and showcases computer graphics techniques.

Unfortunately, due to numerous unforeseen circumstances, project had to undergo several plan changes. The changes and reasons behind them are outlined in more detail in the beginning of the **Chapter 4** and in the **Chapter 6**. As a result the scope of the project was reduced to a research and demonstration of the computer graphics techniques without engine implementation.

1.2 Objectives

There were two different sets of objectives for this project.

The first one included the game engine software engineering aspect and the other one covered scientific part of the project related to research and implementation of techniques.

Game engine:

- Examine modern game engine core functionality and editor UIs.
- Choose which genre would be the most appropriate to demonstrate aforementioned techniques and examine implications of such choice when designing the engine.
- Implement each module of the whole system separately and produce a game engine by enabling interaction between these modules.

Implemented techniques would be seen as the sub-modules of the engine's component system module or simply components.

Techniques and effects:

- Atmospheric scattering.
- Volumetric clouds.
- Volumetric light, god rays, sun shafts.
- Rainbows.

- Halo, parhelion, sun dogs.

In the end, a demo capable of showcasing these effects would be produced based on the implemented game engine.

After project type shifted from group to individual, only objectives listed under “Techniques and effects” were considered. To reflect this change, the report’s body has two big sections with similar chapters. The first one covers the part related to Game Engine development prior to project type shift and the second describes the process after the shift that involved individual research and implementation of computer graphics techniques.

1.3 Deliverables

The project has two deliverables:

- Source code of a software that demonstrates described effects in action, written in C++ and OpenGL 4.6 optimized to run on Windows 10 and tested on Nvidia RTX 2060 Super.
- A report that gives an overview of the development process from idea to software and a thorough explanation of the solutions used to achieve a result shown in the demo application.

1.4 Relevance to degree

Despite this project being significantly lesser in scope than it was originally intended it still has plenty of room to demonstrate the ability to research previous work, seek out a solution to complex problems never encountered before and study new techniques applied in the modern day commercial software that relies heavily on high performance graphics.

Due to occasional plan changes this project also gave an opportunity to show the importance of having a flexible plan that does not sacrifice strictness of milestone deadlines and is easy to adapt to most circumstances.

Section I
Game Engine

Chapter 2
Background Research

Since game engine needs to be highly modular to include various features described in **1.2 Objectives** there should be a list of separate key systems that are required for it to be able to produce a demo with some gameplay.

At their core, flexible game engines like state of the art Unity (16) and Unreal Engine (18) are most often built around an architectural pattern called Entity-Component-System (ECS), which is mainly used to describe game objects (Entities) and characteristics that define them (Components).

For example, entity player could have following components:

- Transform, which defines its position in the world with Model matrix.
- Camera, which defines virtual camera with View-Projection matrix and a resource to which it renders.
- Script, which defines controls that utilize user hardware input.

Such components, which are assigned to entities, should be able to communicate with game engine subsystems or also called support systems which provide interface to external and in-game resources(17).

Key support systems:

- Scene manager – provides an interface to all game objects and their assigned components in the scene, and controls main loop.
- Resource manager – provides an interface to all loaded resources and manages import of any external assets.
- Start up/Shut down manager – loads resources in and out of the memory.
- Human interface device manager – manages hardware through which user interacts with the game.
- Project manager – saves and loads states of the project, which is defined by a scene with a list of serialized game objects and their components..
- Editor – provides a user interface to edit the project – create game objects, assign components, tweak parameters and resources.

Chapter 3
Project Management

3.1 Methodology

The choice of correct methodology is extremely important since it will be reflected on the ability to overcome difficulties in case something goes wrong be it planning, initial assumptions or external issues.

3.1.1 Options and comparison

Agile methodology is very flexible and this trait could be considered both its strength and weakness. By applying agile methodology it is easy to adapt project to any circumstances thanks to its iterative nature, where each iteration is self-contained. The drawback of this option is that it is hard to predict what exactly would be the end result after the final deadline. This makes it impossible to set exact requirements and produce fair expectations. On the other hand, it opens up an opportunity to receive continuous feedback on implemented features as at the end of each iteration a fully working and tested product is produced.

Although waterfall methodology is not able to guarantee a working piece of software at the end of each of its stages to receive continuous feedback, loses flexibility of making changes to already implemented parts, and lacks control over the project during the course of development, it has a clear vision of the end goal, which helps keeping track of the progress as more stages are finished. However choosing waterfall methodology still involves some degree of risk since all set deadlines should be respected. If deadlines are missed, developers are unable to keep up the pace and there is no additional time left to spare, in the best case the testing phase is sacrificed resulting in a significant drop of quality, while in worst case developers are left with no product to demonstrate.

3.1.2 Adopted model

At first, a safer approach of a waterfall methodology was chosen so that it would be easier to track down progress done to the end goal. This also provided an easier way to assign responsibilities, so that each group member would be sure that everyone does their part and the workload is spread fairly.

3.1.3 Planning for the group project

The plan for the group project mostly described the milestones necessary to implement a fully-fledged game engine, which were outlined in **Chapter 2 Background Research**. The workload in a group was spread differently for each contributor’s semester due to different study time allocation. Since balance of the time, that could be allocated to the project, of one of the contributors was shifted towards second semester it was decided that most of the

boilerplate would be finished by the end of the first semester while most of the second semester time would be allocated to implementing core game engine systems and additional ideas dependant on the type of demo that would demonstrate the ability to conduct a research on a complex topic.

Possible demo features:

- Physics and collision mechanism.
- Procedurally generated terrain.
- Mesh optimization and level-of-detail algorithm.
- Skybox and atmospheric effects.
- Realistic lighting.
- AI.
- Networking.

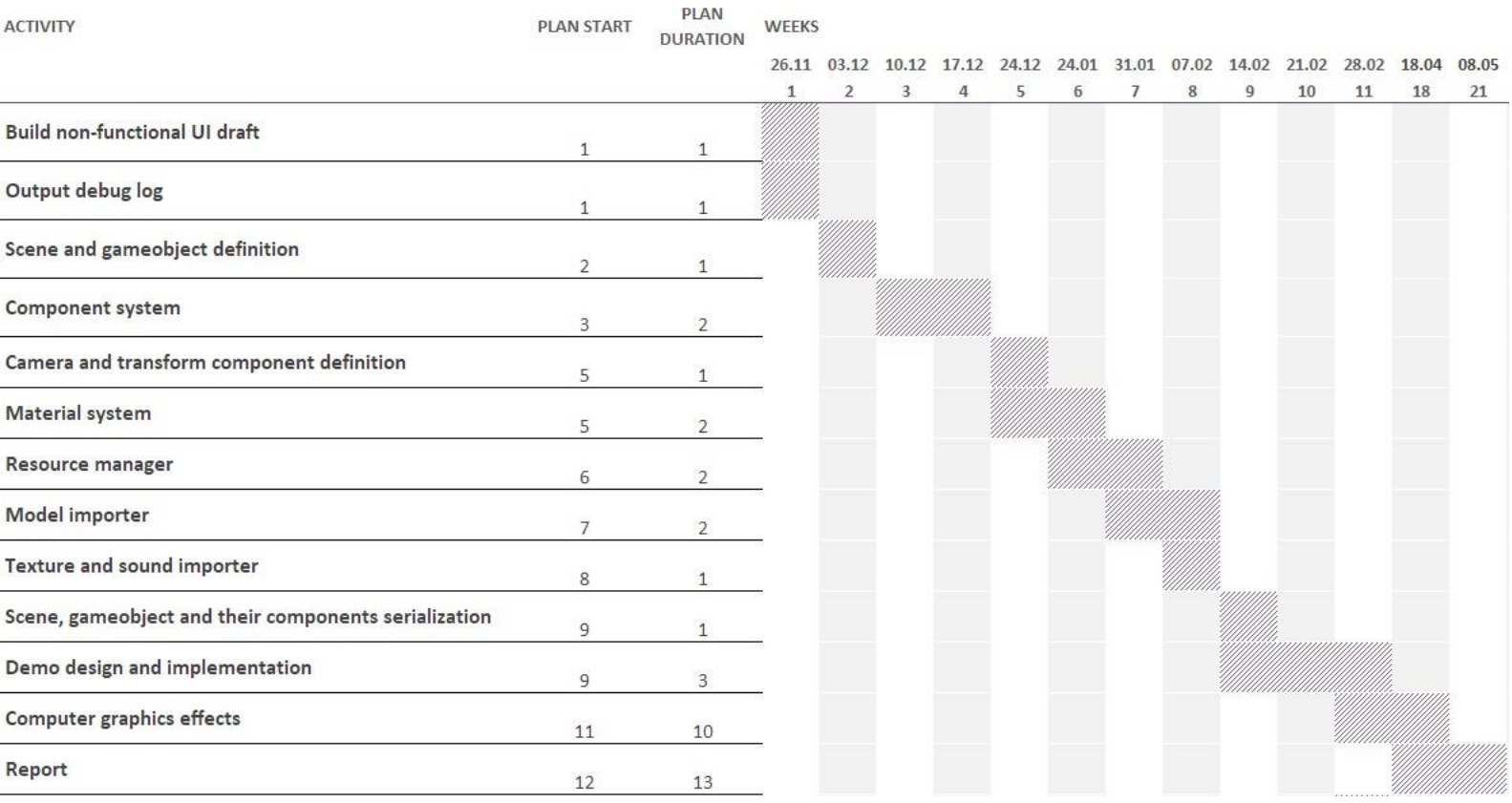

Figure 3.1 First person game engine development plan

Chapter 4
Software Overview

Despite everything was set for practical part, only first week of the plan was finished due to author's health issues arisen in the beginning of the Christmas holidays with subsequent partner withdrawal. This event in result turned group project into individual one.

4.1 Design

4.1.1 UI

Prior to plan changes I was able to build a game engine editor draft. It is fairly simple yet still it is worth mentioning since its creation was a part of the process.

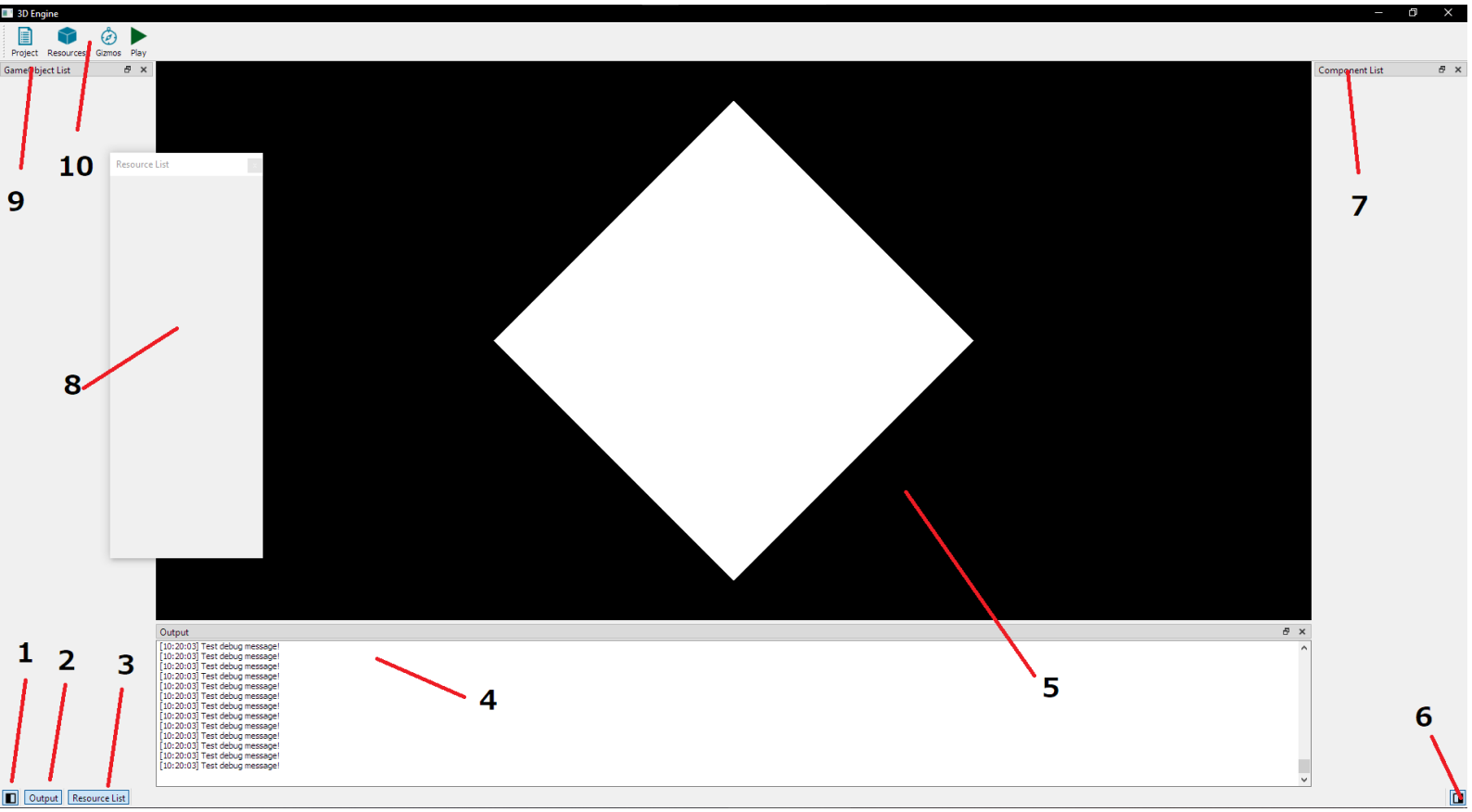

Figure 4.1 Game engine editor user interface

1. Show or hide game object (entity) list button.
2. Show or hide log output button.
3. Show or hide resource list button.
4. Output – shows everything sent to the log from code. Is useful for debugging purposes.
5. Scene interaction panel, where scene comprised of game objects is seen.
6. Show or hide component list.

7. Component list – a list of all supported components that are assignable to game objects.
8. Resource list – a list of all imported resources that can be assigned to components that support such resources, e.g. textures, sounds, models.
9. Game object list – a list of game objects (entities) in the scene.
10. Toolbar that contains several elements from left to right:
    10.1. Project for saving or loading.
    10.2. Resources for game object creation and resource importing.
    10.3. Gizmos to enable or disable different visual cues and widgets in scene interaction panel.
    10.4. Play to start the demo.

Each panel except for the central, toolbar and statusbar is draggable for convenience. If it is docked, it detaches and becomes a floating panel just like resource list (number 8 as seen on a screenshot). Then it is possible to attach it back by dragging it to one of the edges of the window or stack it with another panel and switch between them using tabs. This gives a good amount of customization options for user to choose from. It is also possible to hide panels to increase the size of scene interaction panel.

4.1.2 Controls

Since only UI draft was produced, controls were subject to change which happened and is described later in **Controls**.

Camera's yaw and pitch are controlled using mouse the same way it is done in First Person Shooters. Moving mouse up and down will control pitch, while left and right will control yaw.

Keyboard controls:

- W – Move camera forwards.
- S – Move camera backwards.
- A – Strafe left.
- D – Strafe right.
- R – Move camera up.
- F – Move camera down.
- E – Toggle mouse/keyboard controls.

4.2 Development decisions

4.2.1 Tools

The application is written using C++ language, which is a standard choice when dealing with high performance graphics.

Microsoft Visual Studio 2019 IDE with ReSharper C++ addon provided an immense help with debugging and refactoring of C++ code. Despite the code working seemingly correct I was able to pinpoint at least two bugs just by taking into account code style advices.

4.2.2 Libraries

4.2.2.1 Graphics API

There are at least four well-known and widely used modern APIs:

- OpenGL
- DirectX 11/12
- Vulkan
- Metal

Since Metal is only available on Apple products and I don't have one to work on, it was clearly not an option.

Similarly both DirectX 11 and its low-level low-overhead version DirectX 12 are only available on Windows platform, apart from it being not cross-platform, the author had no previous experience with it, so it made no sense to pick it up to be a part of this project.

Another option apart from modern OpenGL would be low-overhead low-level platform-independent high-performant Vulkan. While it is a viable option, drawbacks would outweigh benefits in author's case. Since author had little experience with it and is not confident enough using it, it would only slow him down and snatch the focus of the project from techniques implementation to making an optimized and fast Vulkan renderer.

This makes OpenGL the best possible choice. It is a cross-platform API, author is fairly familiar with it and it is not as low-level and complicated as Vulkan, Metal or DirectX 12.

4.2.2.2 Qt

To draw something on the screen with OpenGL a context should be provided. Also, to use modern OpenGL (>3.0) calls its functions' pointers should be loaded. Qt does this and many more things. It can be used to create user interfaces, provides a sufficient maths library and input handling, and has some media resource management classes that could be used to load external resources.

Section II
Natural Phenomena

Chapter 5
Background Research

Most atmospheric effects that demonstrate natural phenomena could be described using radiative transfer models, known to be not only exceptionally complex on paper but also extremely difficult to compute even offline(3).

At the same time, lighting is an integral part of any scene, which makes outdoor scenes seem especially artificial without proper sun contribution. Realtime photorealistic graphics are a requirement for simulations and videogames that strive for realistic or cinematic experience, mainly flight simulators (6), first person shooters (7), action adventures (8, 9) and role-playing games (10).

Older solutions of the problem relied on non-interactive implementations like skyboxes with photographed textures, billboard sprite-based clouds, fog, aurora borealis, rainbows and any other effect that is easier to produce by artists than to physically approximate on the fly. Such solutions reduced tweaking capabilities for someone composing a scene by restricting environments and effects to only static options.

Advancements in computer graphics and hardware of the last decade led to many new feasible ways to integrate photorealistic dynamic lighting and atmospheric effects into modern applications by approximating physical models of light behaviour, some of which even after several years are still considered state of art(4, 5).

During this project two separate yet related major topics were researched. The first one is rendering correct atmosphere colours both from inside and outside the atmosphere in realtime and the second is generating and rendering plausible shapes of clouds.

5.1 Density volume rendering

All effects listed in **1.2 Objectives** are a type of optical phenomena produced as a result of light scattering across the participating media, which means that it requires tracing a light path through such media defined by specific physical properties that would determine the amount of light out-scattered and in-scattered at different wavelengths. Apart from that, such properties could be non-uniform throughout the volume, neither homogeneous nor isotropic. This requires sampling data along both the initial path and scattering paths, which increases complexity exponentially when multiple scattering is involved.

That is why as much computation in data sampling logic as possible should be moved to offline. Of course such solution has its drawbacks since with each precomputed part dependant on particular parameters the ability to modify these parameters in real-time is lost as well as memory trade-off increases.

James F. Blinn (11) discussed these issues in his work in 1982 which pioneered in the field of rendering volume data by simulating light interaction with small particles. His work is a compilation of methods and ideas which form basis of every subsequent research that deals with rendering of more specific phenomena like atmosphere and clouds.

5.2 Atmospheric scattering

Atmospheric scattering is a very important atmospheric effect due to its significant contribution to outdoor lighting and earth irradiance as it is seen from space. Manifestation of atmospheric scattering in everyday life includes blue sky, red or orange sunset, blue tint on distant objects and haze.

The main problem behind photo accurate rendering of atmospheric scattering effect in real time is the same as with the most volumetric light scattering algorithms, the trivial solution is extremely slow, yet trying to fake it is easily noticeable.

5.2.1 Related work

The first effort at tackling the subject of atmospheric scattering approximation when rendering earth using real physical data was a research carried out by Nishita et al. (1) in 1993. The work was based on assumption that the earth is spherical and that sunlight is parallel. This made it possible to store optical depth of a ray to the sun at any point in the atmosphere in a look up table

Aforementioned research started a trend of treating atmosphere as a medium consisting of air molecules (for Rayleigh scattering) and aerosol particles (for Mie scattering) and exploiting their physical properties for approximation of light interaction with them when rendering the atmosphere(2, 12, 13). There was an incredible amount of research done based on work by Nishita et al. Most of the state of the art methods including some early attempts were briefly discussed and compared in Eric Bruneton’s (5) survey in 2016.

One is a way presented by Sean O’Neil (13) in 2005, where he proposed a real-time solution of single atmospheric scattering problem without any precomputed tables with scattering integral being completely numerically integrated with a small amount of samples on the fly. Key point is a new way to parametrize samples of the optical depth using view-point altitude from the planet’s surface and sun-zenith angle. The benefit of such approach is that it allows to derive optical depth between any two points in the atmosphere simply by subtracting the

sample value for ray segment that goes from the end point to the top atmosphere boundary from the one that goes from the start point in the same direction.

Schafhitzel et al. (12) went further in their 2007 work by precomputing single scattering into a three dimensional look up table by including a third parameter – view-zenith angle.

5.2.2 Algorithm by Bruneton et al. (2)

Bruneton et al. (2) method, which was implemented in this project builds upon ideas presented in aforementioned works while adding support for precomputed multiple scattering.

5.2.2.1 Rendering equations

The main equation to be solved represents the radiance of light L that is received at specific position x, from direction v with sunlight coming from s:

$$L(x,v,s) = L_0(x,v,s) + R[L](x,v,s) + S[L](x,v,s)$$

Where L_0 represents the direct light, R is the reflected light and S is the inscattered light.

Direct light L_0 is the light coming from sun directly in a straight ray.

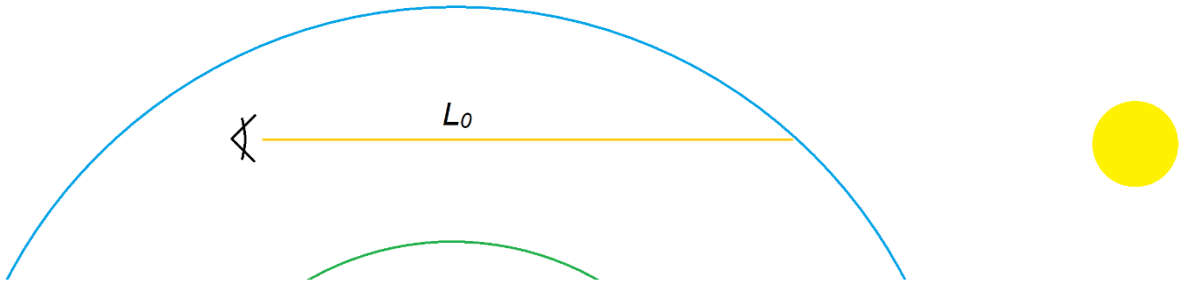

Figure 5.1 Direct light

In case sun is not in the direct line of sight L_0 equals 0, otherwise it is extremely trivial to evaluate and it's equation consists only of two terms:

$$L_0(x,v,s) = T(x,x_0)L_{sun}$$

L_{sun} represents sun radiance, which could be some constant value. T on the other hand is the first integral term that makes computation of atmospheric scattering in real time incredibly complex:

$$T(x,x_0) = exp\left(-\int_x^{x_0} \sum_{i\in\{R,M\}} \beta_i^e(y)\,dy\right)$$

It is the transmittance of the light through the atmosphere due to Rayleigh and Mie extinction factors β^e (absorption and scattering) of the light along the line segment at specific sample

points, defined by two points x and x_0, which in this case are positions of the eye and the end of the line of sight defined by some occluder like planet surface or top atmosphere boundary respectively. The part of equation that is being summed is previously mentioned optical depth.

The extinction factors for both Rayleigh and Mie scattering are derived the following way:

$$\beta^e = \beta^s + \beta^a$$

$$\beta^s(h, H, \lambda) \approx \beta^s(0, H, \lambda) \exp\left(-\frac{h}{H}\right) \quad (14)$$

$$\beta_R^e \approx \beta_R^s \qquad \beta_M^e \approx \frac{\beta_M^s}{0.9}$$

Both extinction terms are approximated from scattering factors. Scattering factor is dependent on the altitude h, density height scale H, which is unique for each type of scattering, and wavelength λ in case of Rayleigh scattering. It is approximated from scattering factor at sea level which exponentially decreases with increase in altitude.

Reflected light R is the light reflected off the planet's surface.

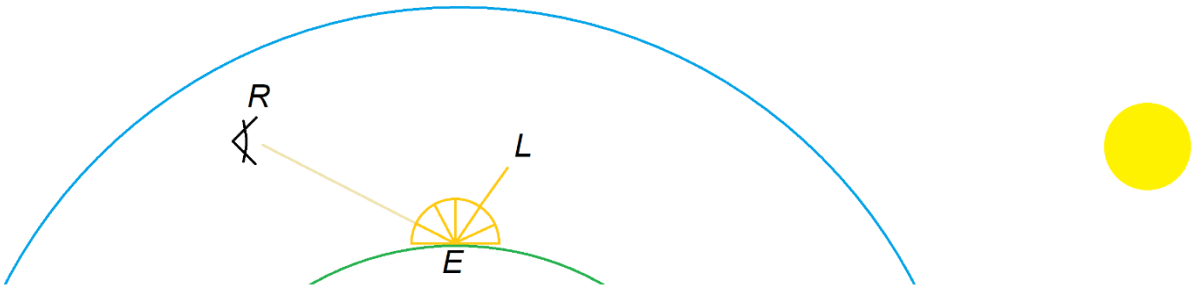

Figure 5.2 Reflected light

While there were attempts to approximate transmittance analytically, approximating reflected light is practically impossible to do accurately if taking multiple scattering into account:

$$R[L](x, v, s) = T(x, x_0)\frac{\alpha}{\pi}E[L](x_0, s)$$
$$E[L](x_0, s) = \int_{2\pi} L(x_0, \omega, s)\omega \cdot n(x_0)d\omega$$

where α is the albedo of the surface and n is the normal. The R term consists of integral over a set of directions in a hemisphere local to the point of reflection x_0. What makes it even harder to compute is the recursive call to the radiance equation L. If ray doesn't hit any surface, i.e. x_0 is a point on the top atmosphere boundary, R evaluates to 0.

Inscattered light S is the contribution of scattered light due to the small particles in the atmosphere in the viewer's direction.

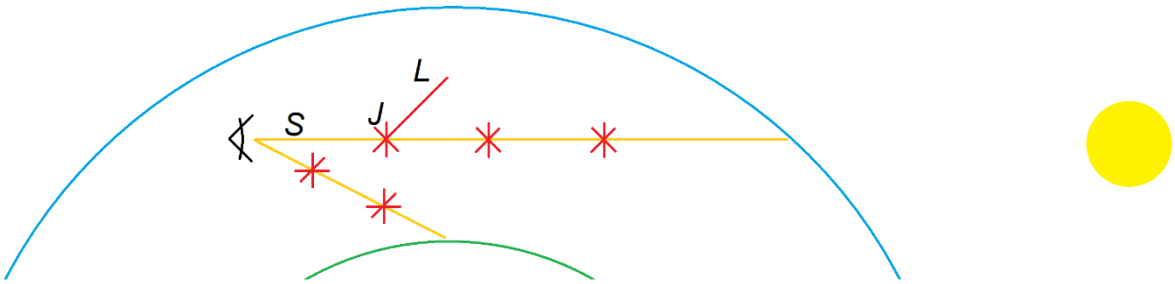

Figure 5.3 Inscattered light

Same as with reflected light, the computation of inscattered light increases in complexity due to its recursive nature. In addition the recursive light equation call is inside a nested integral which makes straightforward solution totally out of the question:

$$S[L](x,v,s) = \int_{x}^{x_0} T(x,y)J[L](y,v,s)\,dy$$

$$J[L](y,v,s) = \int_{4\pi} \sum_{i\in\{R,M\}} \beta_i^s(y)P_i(v\cdot\omega)\,L(y,\omega,s)d\omega$$

The amount of inscattered light is dependent on the angle θ between the incident ray and view direction. This dependency is defined by a phase function P which is unique for each type of scattering due to different asymmetry factor $g \in [-1;1]$ which defines angular distribution of the scattering. In this algorithm an analytical approximation of Henyey-Greenstein phase function by Cornette-Shanks (18) is used:

$$P(\mu) = \frac{3(1-g^2)(1+\mu^2)}{8\pi(2+g^2)(1+g^2-2g\mu)^{\frac{3}{2}}}$$

$$\mu = cos\,\theta$$

In case of P_R, g is equal to 0, which yields a uniform distribution and much simpler equation. For P_M, however, g evaluates to approximately 0.76, which results in a strong forward direction of scattered rays.

5.2.2.2 Algorithm overview

Algorithm distinguishes terms that involve zero (no) scattering, single scattering (one incident ray per sample point directly to the sun) and multiple (at each sample point sum of higher orders of) scattering which yields the following rendering equation:

$$L = L_0 + L_1 + L_*$$

$$L_1 = R[L_0] + S[L_0]$$

$$L_* = \sum_{n>1} R[L_{n-1}] + S[L_{n-1}]$$

Since direct light L_0 and reflections with zero scattering R_0 cannot be precomputed without approximations that will noticeably degrade image quality and easy to compute in real-time accurately without much performance impact, the precomputations are only done for terms that involve integrals, i.e. T, $S[L_0]$ and L_*.

The precomputations are done for all possible altitudes, view and sun directions assuming that both bottom and top atmosphere boundaries are ideally spherical, so the end point of the view direction is always hitting one of the two. To account for occlusion effects, e.g. due to terrain, several tricks are suggested to calculate transmittance and scattering between any two points:

- Idea suggested by O'Neil (13) in 2005 is reused, this time to calculate the transmittance complete with exponential term and added up optical depths for each type of scattering:

$$T(x, x_0) = \frac{T(x, v)}{T(x_0, v)}$$
$$v = \frac{x_0 - x}{\|x_0 - x\|}$$

- Bruneton et al. (2) suggest calculating single scattering $S[L_0]$ term only for points that are not in shadow, since other have their integrand evaluated to 0.

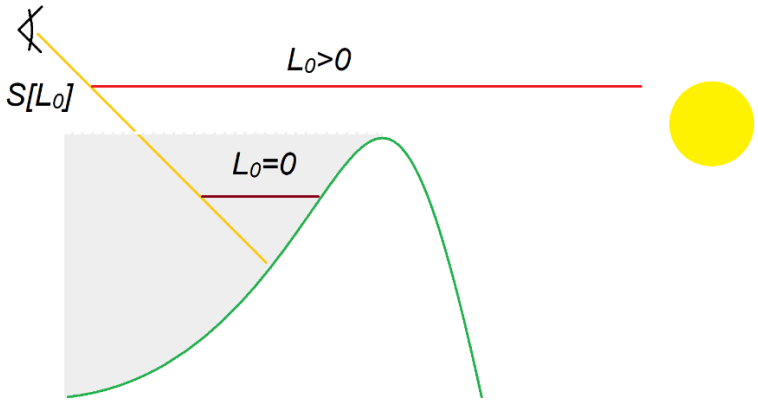

Figure 5.4 $S[L_0]$ term going through shadow

If x_s is a point at which shadow starts along the ray then $\bar{S}[L_0]$ to the shadow point is calculated the following way:

$$\bar{S}[L_0](x, x_s, v, s) = S[L_0](x, v, s) - T(x, x_s)S[L_0](x_s, v, s)$$

- Occlusion effects for multiple scattering $\bar{S}[L_*]$ are approximated the same way since accounting for occlusion of all scattering paths is impractical. The result is plausible since error produced for lighting inside and outside of the shadow is small and almost negates itself considering how much multiple scattering contributes to overall accumulated light. Due to this, it is possible to accumulate both single and multiple scattering $S[L]$ together in one precomputed table.

- Reflected light in multiple scattering $\bar{R}[L_*]$ is also considered by approximating occlusion due to ground's tangent plane, where $n(x_0)$ is the normal of a sphere and $\bar{n}(x_0)$ is the normal of a planet's actual terrain. This gives a modified formula:

$$\bar{R}[L](x,v,s) = \frac{1 + n(x_0) \cdot \bar{n}(x_0)}{2} T(x,x_0) \frac{\alpha}{\pi} E[L](x,s)$$

5.2.2.3 Precomputation

The transmittance T, irradiance (light accumulated at specific point) E and inscattered light S are precomputed in three tables $\mathbb{T}$, $\mathbb{E}$ and $\mathbb{S}$ with three intermediate tables used exclusively for multiple scattering precomputation, $\Delta\mathbb{E}$, $\Delta\mathbb{J}$ and $\Delta\mathbb{S}$. Bruneton et al. (2) in their paper suggest the following algorithm for precomputation of the aforementioned tables:

$$\begin{aligned}
&\mathbb{T}(x,v) \longleftarrow T(x,x_0)\\
&\Delta\mathbb{E}(x,s) \longleftarrow E[L_0](x,s)\\
&\Delta\mathbb{S}(x,v,s) \longleftarrow S[L_0](x,v,s)\\
&\mathbb{S}(x,v,s) \longleftarrow \Delta\mathbb{S}(x,v,s)\\
&\textbf{for } i \longleftarrow 1 \textbf{ to } i < n_{orders}\\
&\textbf{do} \begin{cases}
\Delta\mathbb{J}(x,v,s) \longleftarrow J[T\frac{\alpha}{\pi}\Delta\mathbb{E} + \Delta\mathbb{S}](x,v,s)\\
\Delta\mathbb{E}(x,s) \longleftarrow E[\Delta\mathbb{S}](x,s)\\
\Delta\mathbb{S}(x,v,s) \longleftarrow \int_x^{x_0} T(x,y)\Delta\mathbb{J}(y,v,s)\,dy\\
\mathbb{E}(x,v,s) \longleftarrow \mathbb{E}(x,v,s) + \Delta\mathbb{E}(x,v,s)\\
\mathbb{S}(x,v,s) \longleftarrow \mathbb{S}(x,v,s) + \Delta\mathbb{S}(x,v,s)
\end{cases}
\end{aligned}$$

They also extend ideas introduced by O'Neal (13) and Schafhitzel et al. (12). By exploiting spherical properties of the atmosphere the position component x reduces to altitude r, view direction v to view-zenith angle cosine μ and sun direction to sun-zenith and sun-view angle cosine, μ_s and ν respectively. This makes T and E to depend only on two parameters r and μ while *S* and *J* depend on four r, μ, μ_s and ν. This is very convenient since it allows to precompute all the data using corresponding $\mathbb{T}$, $\Delta\mathbb{E}$ and $\mathbb{E}$ 2D textures and $\Delta\mathbb{J}$, $\Delta\mathbb{S}$, and $\mathbb{S}$ 4D textures.

Parametrization for $\mathbb{T}$, $\mathbb{E}$ and $\Delta\mathbb{E}$ is trivial. It is a linear mapping for both coordinates:

$$r \in [R_g, R_t] \; and \; \mu, \mu_s \in [0,1] \text{ to } [0,1]^2$$

Bruneton et al. (2) point out that such simple parametrization won't work for $\Delta\mathbb{J}$, $\Delta\mathbb{S}$ and $\mathbb{S}$ unless the texture will be big enough to produce enough detail for critical regions. Hence, an improved mapping is suggested:

$$u_r = \frac{\rho}{H}$$

$$\rho = \sqrt{r^2 - R_g^2} \qquad H = \sqrt{{R_t}^2 - R_g^2}$$

ρ is the distance to horizon at specific altitude and H is the distance to horizon at the top atmosphere boundary (max possible distance). Such altitude parametrization dedicates more memory to details at lower altitude where atmosphere is much denser.

$$u_\mu = \begin{cases} \frac{1}{2} - \frac{\left(-r\mu - \sqrt{\Delta}\right)}{2\rho} & if\ r\mu < 0\ and\ \Delta > 0 \\ \frac{1}{2} + \frac{\left(-r\mu + \sqrt{\Delta + H^2}\right)}{2\rho + 2H} & otherwise \end{cases}$$
$$\Delta = (r\mu)^2 - \rho^2$$

where Δ is a discriminant of a quadratic equation with distance to the intersection of a ray with the planet's surface as a root. The first formula is for rays intersecting with the planet's surface, while the second one is for rays intersecting with the atmosphere top boundary, hence $\Delta + H^2$ in a discriminant. Solution for distance to intersection is enclosed in braces. The discontinuous mapping of the view angle accounts for discontinuity of the length of the view ray near horizon, mitigating issues that arise with linear interpolation of two parts.

$$u_{\mu_s} = \frac{1 - e^{-3\mu_s - 0.6}}{1 - e^{-3.6}}$$

This ad hoc mapping dedicates more space to sun-zenith angles at horizon completely ignoring angles at which sun no longer provides any contribution. Works properly only with Earth and Sun parameters.

$$u_\nu = \frac{1 + \nu}{2}$$

and a simple linear mapping for view-sun angles.

It is suggested to use a 3D $\mathbb{S}$ texture with 3rd coordinate being a mix of both u_{μ_s} and u_ν with manual linear interpolation for u_ν.

Applying phase functions directly during precomputation of the single scattering results in graphic artifacts due to the limited resolution of each parameter in a 4D texture caused by memory restrictions. Since the problem is much more prominent for single Mie scattering, it is reasonable to store single Mie scattering separately. This gives two separate datasets:

$$C_M = \frac{S_M[L_0]}{P_M}$$
$$C_* = \frac{S_R[L_0] + S[L_*]}{P_R}$$

Bruneton et al. (2) provide a way to approximate single Mie scattering green and blue channels from only red channel without losing any visual fidelity by exploiting a proportionality rule between single Mie and Rayleigh scattering:

$$C_M \approx C_* \frac{C_{M,r}}{C_{*,r}} \frac{\beta_{R,r}^s}{\beta_{M,r}^s} \frac{\beta_M^s}{\beta_R^s}$$

This makes it possible to have only one $\mathbb{S}$ texture that has four channels. Three for C_* and one for $C_{M,r}$.

5.2.2.4 Rendering

Rendering of the atmospheric scattering is done by sending a ray through each pixel of the screen and computing the result of the lighting equation described in the beginning of the **5.2.2.1 Rendering equations**. The terms that are evaluated in real-time, specifically L_0 and R_0 rely on the transmittance data stored in $\mathbb{T}$. Then $\mathbb{E}$ provides data for E term of $R[L_*]$ and $\mathbb{S}$ gives C_* and C_M that must be multiplied by phase terms of corresponding single scattering types, the sum of the result gives $S[L]$ term of the lighting equation.

5.3 Volumetric clouds

Generating and rendering volumetric clouds in real time employs a slightly different approach from the one used for rendering atmospheric scattering. The main difference is due to the complexity of generating a plausible shape of the cloud which is tied to the distribution of small water vapor and ice particles.

Approach mostly used in modern applications involves generating various noise types and its combinations and applying different filters to achieve something that would resemble a cloud. Rendering is also more convoluted since it is no longer possible to shift light interaction calculations to offline because view-rays are now passing through volumes of noisy, almost random data approximating which leads directly to the loss of detail.

5.3.1 Related work

There are two types of very popular noises which are usually used for cloud shape generation.

The first one is by Ken Perlin (19, 20) introduced in his 1985 paper and improved in 2002 paper commonly known as Perlin noise. The idea is very simple:

- Consider every *corner* point with integer coordinates in xyz space.
- For each point generate a pseudorandom *gradient* vector.
- For each *input* coordinates of some point, calculate vectors to that point from surrounding 8 corner points.
- Calculate dot product of each *gradient* vector and vector to *input* pair.
- Perform trilinear interpolation of resulting values on each of the *corner* points by using *smootherstep* function introduced in improved algorithm and a fraction part of the *input* coordinates as x(20).

$$smootherstep(x) = 6x^5 - 15x^4 + 10x^3$$

The second is Worley noise introduced by Steven Worley (21) in 1996, which is even simpler than Perlin noise:

- Consider a grid of uniformly spaced cubes with integer coordinates in xyz space.
- For each cube generate a *feature* point with pseudorandom coordinates within cube.
- For each *input* coordinates of some point, determine to which cube it belongs.
- Evaluate distance from input point to each point of 9 surrounding cubes.
- Store closest distance as a result.

Apart from various noises there is another approach commonly integrated in cloud shape generation called fractional Brownian motions (fBm) (23). It is used to seamlessly combine noises of different frequencies. It works by adding with each iteration and then normalizing noise of gradually increasing frequency and decreasing amplitude.

5.3.2 Algorithm by Schneider et al. and Nubis system (22)

The algorithm by Schneider et al. is one of the most advanced modern approaches to rendering interactive volumetric clouds. It is used in modern video games like Nubis system in Horizon Zero Dawn (9) and Red Dead Redemption 2 (10).

It considers many types of clouds, but sticks to rendering volumetrically only lower layer and cumulonimbus clouds. As all the other types could be rendered using 2D textures.

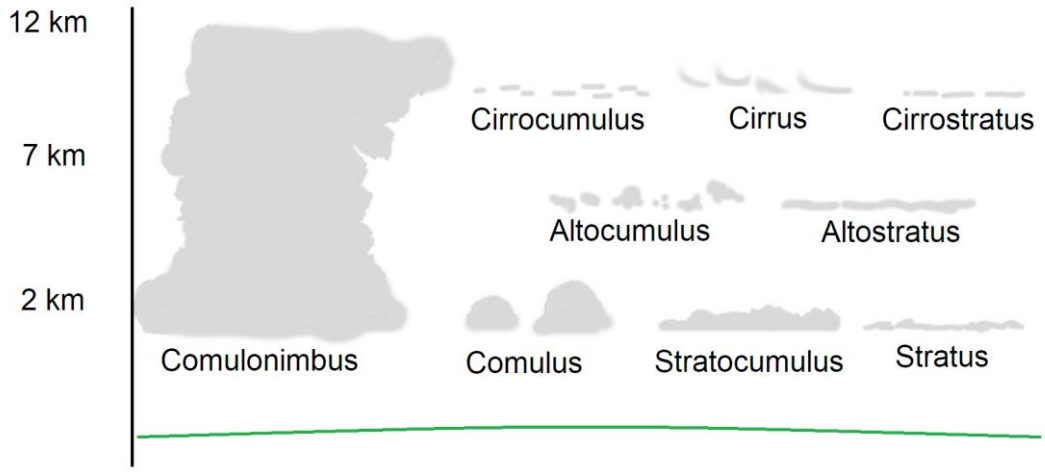

Figure 5.5 Cloud types

Schneider et al. (22) start with noise generation. For cloud shape generation are considered noises that would produce billowy, puffy shapes, hence Perlin and inverted Worley noise. In addition to that a new type of noise is introduced called Perlin-Worley noise which is a combination of two. Schneider et al. (22) suggest generating it the following way:

- Generate Perlin noise.

- Generate Worley noise, invert it and use it in fBm to achieve a fractal cauliflower-like shape.
- Combine two by remapping Perlin noise from its old range with lower boundary being values from Worley fBm.

$$PerlinWorley = NewMin + \frac{(Perlin - WorleyFBM) * (NewMax - NewMin)}{OldMax - OldMin}$$

The result is a Perlin noise with Worley noise billowy shapes in low-density regions.

In the end using described approaches two 3D textures should be generated. One higher resolution with four channels for Perlin-Worley and three octaves of low-frequency Worley noise fBms and a second one lower resolution with three channels each for different octaves of high-frequency Worley noise. Perlin-Worley is then sampled for main shape. Afterwards, three low-frequency Worley noise fBms are combined into one fBm which defines the cloud shape by remapping Perlin-Worley noise using inverted fBm that was just build as the old lower boundary. The high-frequency noises from the second texture are used to add details to the edges of the cloud in the same way, yet they produce wispy shapes.

There are some additional control textures for coverage, precipitation and cloud type which are mainly created by artist:

- Coverage determines the amount of clouds in the sky and is used as a value to remap sampled noise.
- Precipitation controls clouds density and a chance that rain will be produced.
- Cloud type determines which gradient will be used for altitude control.

Chapter 6
Project Management

As soon as the first group project impeding circumstances occurred it became apparent that rethinking of both strategy and planning is necessary due to the change of the scope and focus of the project.

6.1 Updated project management model

The unpredictability of the project forced an adoption of a more flexible approach, resulting in a blend of both agile and waterfall methodologies with more traits taken from agile model:

- Several ideas were brought up that would mark the topics of each upcoming agile iteration.
- Each iteration required its stages to be planned prior to commencing in the way it is done for the waterfall approach globally.
- In case the project falls short of time, even if only one iteration was done, it would be sufficient to review the carried out research and demonstrate an implementation of idea assigned to this iteration.
- The project still holds a clear vision of what it strives to be and the whole approach is more to akin to polishing rather than adding new features that change the whole picture.
- At any given stage be it the beginning of the new iteration or some stage in the middle of one it is possible to give an update and an estimate of the remaining work.

This hybrid methodology tries its best to preserve clear vision of the end goal while not making dangerous assumptions by delaying the complete planning of each iteration until it actually begins.

6.2 Planning for the individual project

The initial revision of the plan after the project changed its focus was extremely optimistic with the idea to tackle as many visual effects listed in **1.2 Objectives** as time will allow until the writing part will have to begin. Essentially, every bullet point was listed as a separate agile iteration.
Later on, after implementing a simple single atmospheric scattering algorithm on a CPU, some further research and almost a year of suspension due to medical condition of the author of this report it became evident that even the first taken up effect is an extremely broad and dense topic and implementing an accurate and performant demo would take a lot of time, learning new approaches and building understanding of the modern solution along the way.

The next immediate decision was to spend some time learning how shaders work and doing some practice before trying to implement something much more complex. Since even the simplest solution on a CPU at extremely low resolutions produced results not even close to realtime, the first significant step of the whole project was implementation of the spherical layer volume renderer by transferring the CPU solution to GPU which took roughly 3 weeks.

This marked the point at which the final plan could be produced with atmospheric scattering being the main focus and all the time that is left until the writing part allocated to cloud rendering. All other effects were deemed as unfeasible due to time constraints.

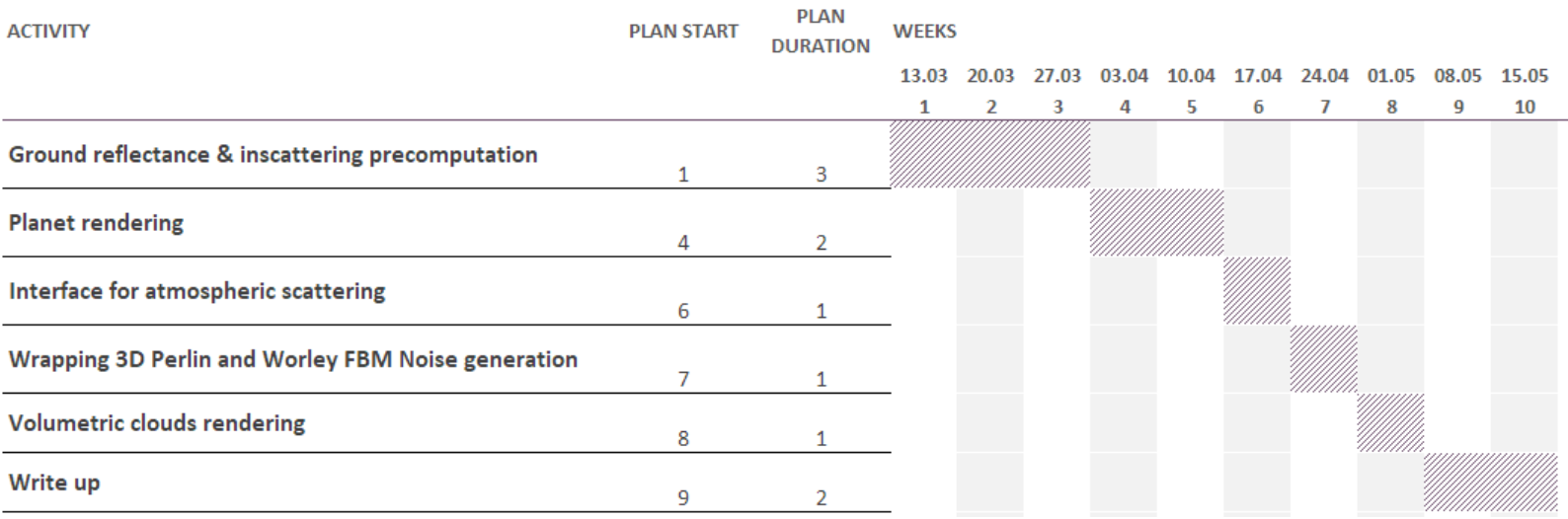

Figure 6.1 Natural phenomena final plan

Chapter 7
Software Overview

7.1 Design

As soon as the application starts it auto-fills parameters for a photorealistic earth atmosphere and precomputes all data. It takes very little time so it should not be noticeable. Just as the application finishes setup it lets the user to control the camera and change parameters at will which are applied with a press of the corresponding button.

7.1.1 UI

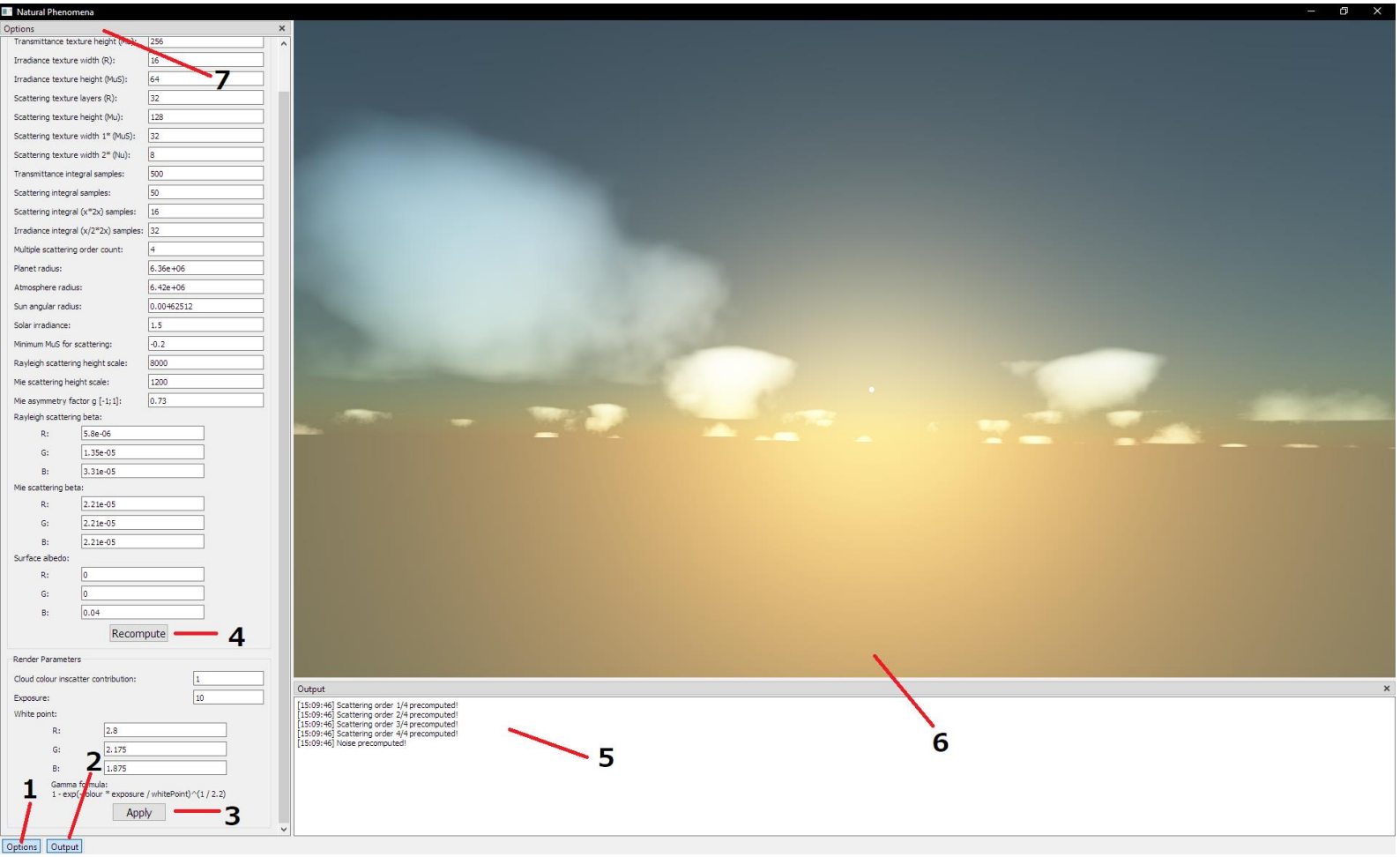

Figure 7.1 Natural phenomena demo screenshot

1. Show or hide parameters list.
2. Show or hide log output.
3. Apply parameters without recomputation of data.
4. Apply parameters and recompute all the textures.

5. Output – shows everything sent to the log from code. Is useful for debugging purposes and outputting performance metrics.
6. Render panel.
7. Tweakable parameter list.

The user interface is a reworked version of what was presented in **4.1.1 UI**. It was modified so that there would be only two non-detachable but hidable panels. One for parameter tweaking and the other one for log output

7.1.2 Controls

The controls are mostly identical to the ones described in **4.1.2 Controls** except for some minor changes.

- Right Mouse Button – Hold when cursor is inside render panel to activate mouse/keyboard controls and release to deactivate them.
- E – Increases sun angle.
- Q – Decreases sun angle.

It is impossible to go through the planet. Whenever user tries to go through the surface he gets pulled out back to 1.5 meters above the surface.

7.2 Development decisions

The decisions for the revised version of this project are mostly the same with explanation of the software structure and a minor addition to the **4.2.1 Tools** which is outlined further.

7.2.1 Tools

Since MS Visual Studio doesn't provide graphics debugger for OpenGL applications I found RenderDoc extremely helpful. One of the most useful features was the ability to see a disassembly of the compiled shader program. Also, I constantly used texture viewer to lookup output values and compare textures directly in different channels.

7.2.2 Software structure

There are seven C++ source files and six header files:

- main(.cpp/.h) – The application entry point.
- Window(.cpp/.h) – Defines OpenGL context window and its interface.
- Widget(.cpp/.h) – Defines Qt OpenGL widget with main rendering loop.
- Shader(.cpp/.h) – Shader compilation and program linker helper class.
- Renderer(.cpp/.h) – Defines actual render calls. Stores OpenGL state data.
- Atmosphere(.cpp/.h) – Defines and controls parameters of atmospheric effects.

Apart from that there is a Shader folder with fifteen GLSL shader files.

- common.glsl – Defines common functions used across various shaders. Header file.
- precomputeVS.glsl – Vertex shader that generates fullscreen triangle using texture coordinates derived from vertex ID. Also passes instance ID in case instanced call for 3D texture was issued to process it in geometry shader.
- instanced3DLayerSetterGS.glsl – Geometry shader that handles instanced draw calls to 3D texture.
- transmittanceFS.glsl – T precomputation fragment shader.
- directIrradianceFS.glsl – $E[L_0]$ precomputation fragment shader. Necessary for first order of multiple scattering. Isn't stored.
- singleScatteringFS.glsl – $S[L_0]$ precomputation fragment shader.
- copySingleScatteringFS.glsl – Copies $S[L_0]$ precomputed data from $\Delta\mathbb{S}$ to $\mathbb{S}$ in a fragment shader.
- firstStepMultipleScatteringFS.glsl – $J\left[T\frac{\alpha}{\pi}\Delta\mathbb{E}+\Delta\mathbb{S}\right]$ precomputation fragment shader.
- indirectIrradianceFS.glsl – $E[\Delta\mathbb{S}]$ precomputation fragment shader.
- secondStepMultipleScatteringFS.glsl – $\int_x^{x_0} T(x,y)\Delta\mathbb{J}(y,v,s)\,dy$ precomputation fragment shader.
- copyIrradianceFS.glsl – Copies $E[\Delta\mathbb{S}]$ precomputed data from $\Delta\mathbb{E}$ to $\mathbb{E}$.
- copyMultipleScatteringFS.glsl – Copies $\int_x^{x_0} T(x,y)\Delta\mathbb{J}(y,v,s)\,dy$ precomputed data from $\Delta\mathbb{S}$ to $\mathbb{S}$.
- cloudNoiseFS.glsl – Generates cloud noise 3D textures in a fragment shader.
- planet.glsl – Main planet and atmospheric effects renderer.
- screen.glsl – Combined shader to draw a fullscreen triangle on a screen with a provided texture.

After starting the application Qt is initialized and the main window with interface is created.

The window begins initializing a chain of dependencies. It provides a context for OpenGL through QOpenGLWidget Widget, which constructs Atmosphere object and in the process of OpenGL initialization constructs Renderer with Atmosphere handle being passed to Renderer. This way Atmosphere parameters are accessible before OpenGL initializes.

Renderer class constructs all the textures and shaders and begins precomputation process during which it reports the amount of time taken to the singleton object LogWindow.

User at any point is able to change both precomputation and rendering parameters. Changing the first ones will initiate a new precomputation process.

Chapter 8
Implementation

Further is described the process of development picking up from the progress done in **Section I**.

8.1 Shader class

Shader class fulfils three purposes:

- Shader source assembly from input files.
- Shader compilation.
- Program linking.

It also has debugging capabilities as it reports errors related to shader loading, compilation and linking. If compilation error occurred it also reports shader type in which error occurred along with the error information.

Shader class is tailored to work with specific types of shaders and sources. It is able to load both separate sources for each shader stage and combined shader source to link a complete program. The requirement for combined shader file is to surround code related to each shader stage in #ifdef <Shader stage> #endif macros, where <Shader stage> could be either VERTEX, GEOMETRY or FRAGMENT. Header that consists of version directive (default target is 460) and a common.glsl source file gets appended to the beginning of each shader stage source.

After linking shader program it is sufficient to call use() method in OpenGL context to activate it.

8.2 Renderer

The renderer has three stages – initialization, precomputation and rendering. Each stage is discussed separately.

8.2.1 Initialization

When renderer is constructed the first thing it does apart from initializing OpenGL functions in current context is saving a handle to atmosphere object that contains all the parameters needed for precomputation and rendering.

Then it compiles and links all the shader programs and stores them as Shader objects.

Once that is done, the framebuffer object is built. It has two colour attachments so that it would be possible to reuse it for all operations be it precomputation into two draw buffers simultaneously or a simple render to the screen. The second colour attachment has a

texture assigned only when needed, because rendering using only one draw buffer, when the second colour attachment is active, results in an error. The render buffer is constructed as well, but it is always detached when rendering off screen, since there is no use to depth or stencil buffers.

Each time precomputation initiates be it at initialization step or later when user changes parameters, every texture must be reallocated.

Must note that Qt is known for corrupting the state by periodically stealing OpenGL context to draw its interface. To work around this there are functions that rebind framebuffer and textures that are required for rendering after precomputations are done.

8.2.2 Precomputation

Precomputations are done in a fragment shaders by rendering a primitive that covers [-1,1] region of normalized device coordinates in X and Y coordinates. This is done by rendering one triangle at 0 Z coordinate large enough to cover this region completely. The idea was proposed by Timothy Lottes (24) in 2009 in his anti-aliasing algorithm called FXAA. A draw call to render one triangle without any vertex data is issued. Positions and texture coordinates for each of the three vertices are generated entirely in vertex shader depending on the id of a vertex.

$$TextureCoordinates \leftarrow ((VertexID \ll 1)\ \&\ 2, VertexID\ \&\ 2)$$
$$Position \leftarrow TextureCoordinates * 2 - 1$$

Table 8.1 Position and texture coordinates in relation to vertex id

$VertexID$	0	1	2
$TextureCoordinates$	$(0,0)$	$(2,0)$	$(0,2)$
$Position$	$(-1,-1)$	$(3,-1)$	$(-1,3)$

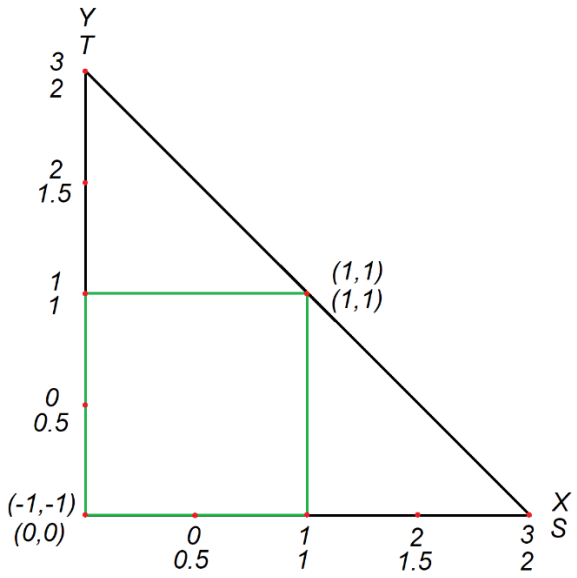

Figure 8.2 Triangle covering [-1,1] region of both XY coordinates

For 3D textures instanced draw calls are used and vertex shader also passes instance id to the geometry shader to determine to which layer current set of vertices belongs and should be emitted to.

There are three textures that must be precomputed for real time atmosphere scattering rendering – transmittance, irradiance and scattering. For each precomputation there are two issues to consider – parametrization and the actual precomputation algorithm.

When precomputing data, that is going to be interpolated across data points, to a texture using fragment shader it is mandatory to render results for values used for parametrization at ranges' ends exactly in the centre of the texel.

Data is going to be precomputed in a fragment shader to a texture, which is going to be interpolated for all intermediate values. Hence, when parametrizing inputs like altitude and angles it is mandatory for inputs at range boundaries to precompute data in edge texels, otherwise they will be extrapolated, which gets more noticeable the lower is the resolution of a precomputed texture. To account for fragment shader always rendering to the $[0.5,0.5]$ coordinates of each texel the coordinate range should be shifted from $[0.5, n-0.5]$ to $[0.0, n-1]$ where n is a number of texels in a row or column. The conversion to normalized coordinates is then done the following way:

$$normalizedCoordinates = \frac{inputCoordinate - 0.5}{n-1}$$

The inverse mapping when reading data back for specific inputs (altitude and angles) is straightforward, just have to note that conversion is done purely for normalized inputs, no texel coordinates involved, since parametrization formulas are applied first.

8.2.2.1 Transmittance

The algorithm starts with transmittance since both other textures are dependent on it. Since transmittance shares properties with scattering instead of using linear mapping for altitude much better results are achieved by utilizing the mapping proposed for scattering textures $\Delta\mathbb{J}$, $\Delta\mathbb{S}$ and $\mathbb{S}$:

$$u_r = \frac{\rho}{H}$$

$$\rho = \sqrt{r^2 - R_g^2} \qquad H = \sqrt{{R_t}^2 - R_g^2}$$

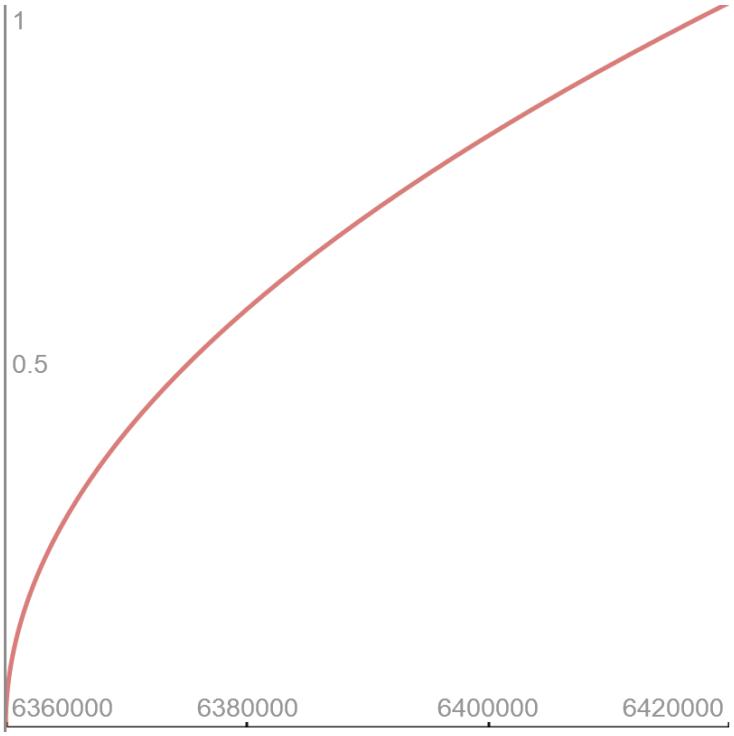

Figure 8.3 Parametrization of altitude with Earth parameters $R_t = 6420\ km$ and $R_g = 6360\ km$

Notice how most of the space is dedicated to lower altitudes.

Along with altitude a different parametrization for angles is used. There is no need to precompute transmittance for points between viewer and planet surface, when it is possible to derive transmittance between these two points using only transmittance to the atmosphere top boundary simply by reversing ray direction and swapping start and end point of the segment based on suggestion by Sean O'Neal (13) as was briefly discussed in the end of **5.2.1 Related work**:

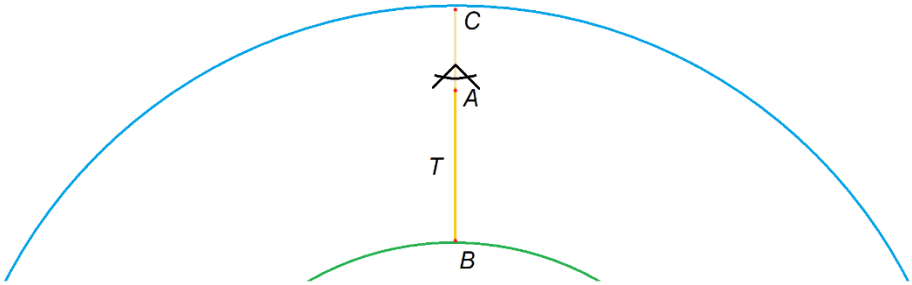

Figure 8.4 Transmittance of a ray intersecting surface of the planet

$$T_{AB} = exp\left(-\int_A^B \sum_{i\in\{R,M\}} \beta_i^e(y)\,dy\right)$$

$$T_{AB} = exp\left(\left(-\int_B^C \sum_{i\in\{R,M\}} \beta_i^e(y)\,dy\right) - \left(-\int_A^C \sum_{i\in\{R,M\}} \beta_i^e(x)\,dx\right)\right)$$

$$T_{AB} = \frac{T_{BC}}{T_{AC}}$$

Hence only angles for view rays that do not intersect planet's surface are considered. The second half of view-zenith angle parametrization for scattering u_μ is doing exactly that. To use it for a whole texture and not only a half, a small modifications must be made:

$$u_\mu = \frac{1}{2} + \frac{\left(-r\mu + \sqrt{\Delta + H^2}\right)}{2\rho + 2H} \quad \frac{1}{2} \le u_\mu \le 1 \quad |\cdot 2$$

$$2u_\mu = 1 + \frac{\left(-r\mu + \sqrt{\Delta + H^2}\right)}{\rho + H} \quad 1 \le 2u_\mu \le 2 \quad |-1$$

$$\bar{u}_\mu = 2u_\mu = \frac{d}{\rho + H} \quad 0 \le \bar{u}_\mu \le 1 \quad d = -r\mu + \sqrt{(r\mu)^2 - \rho^2 + H^2}$$

$$\bar{u}_\mu = \frac{d}{d_{max}} \quad 0 \le \bar{u}_\mu \le 1 \quad d = -r\mu + \sqrt{(r\mu)^2 - r^2 + {R_t}^2} \quad d_{max} = \rho + H$$

d is the distance to the top atmosphere boundary at altitude r with view-zenith angle μ. d_{max} is a maximum possible distance to the top atmosphere boundary at altitude r (it is a length of the ray going to top atmosphere boundary through the horizon point).

But $\bar{u}_\mu$ still doesn't give perfect mapping due to the unused region closer to $\bar{u}_\mu = 0$. To overcome this, the smallest possible distance to horizon at altitude r should be subtracted from both numerator and denominator (25):

$$\bar{\bar{u}}_\mu = \frac{d - d_{min}}{d_{max} - d_{min}} \quad d_{min} = R_t - R_g$$

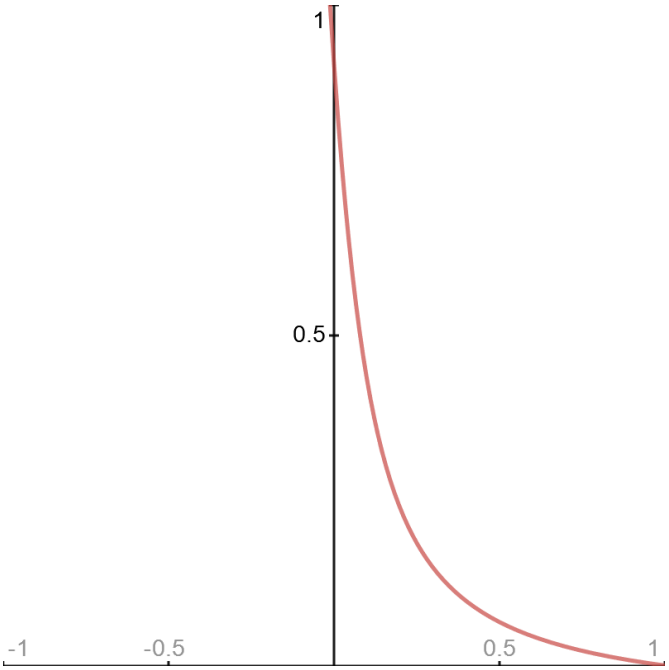

Figure 8.5 Parametrization of view-zenith angle cosine with Earth parameters $R_t = 6420\ km$ and $R_g = 6360\ km$ and altitude $r = 300\ m$

The resulting $\bar{\bar{u}}_\mu$ parametrization is great because it changes dynamically with altitude which allows it to allocate texture space only for data that otherwise cannot be derived, most of it dedicated to critical regions, like those involved in computing light near horizon.

Significant amount of time was wasted to research and fix platform dependent bug in NVIDIA GPUs GLSL implementation which produces wrong results when big float numbers are involved in equations of type:

$$\mathrm{x}^2 - \mathrm{y}$$

The bug (2778487) was pointed out to NVIDIA staff and as of now is still listed as open on NVIDIA bug tracker. The only workaround is to force precise calculations using *precise* qualifier which stops code optimizer from optimizing statements that define the result stored in a variable for which this qualifier was set. Since there are lots of places where variables like ρ and H are calculated which fall into the category of aforementioned equations, setting *precise* for them is the best way to avoid software breaking issues.

The actual precomputation of transmittance is the simplest of all textures. For each texel altitude, view-zenith angle and distance to intersection are determined, which fortunately is the right amount of data to determine the optical depth, which integral is approximated using trapezoidal rule.

Figure 8.6 Precomputed transmittance texture

8.2.2.2 Irradiance

The irradiance parametrization of altitude and sun-zenith angle is linear since its function is smooth (25) and it does not require much precision to get plausible results. This is important because it allows not using the sun-zenith angle parametrization suggested for scattering, which uses ad hoc constant for a specific planet (Earth in case of Bruneton et al. algorithm (2)).

Precomputation of irradiance is done in two separate stages. First direct irradiance is precomputed and stored in $\Delta\mathbb{E}$ for later use in multiple scattering precomputation and then indirect irradiance is precomputed in iteration for each order, which makes use of previous data precomputed for multiple scattering.

The algorithm that precomputes direct irradiance is very simple. After determining altitude and sun-zenith angle it is checked whether sun is above the horizon, is partially set or is completely beyond the horizon depending on the input data. If it is beyond the horizon the contribution is 0. If it is above the horizon the contribution evaluates to solar irradiance by sun-zenith angle cosine and transmittance to the sun from the surface point. If the sun is partially beyond the horizon, then its contribution is approximated ($\alpha \approx cos\,(\frac{\pi}{2} - \alpha)$) (25) in relation to the radiative view factor to a sphere (26), which instead of multiplying the result by sun-zenith angle cosine when sun is completely seen, multiplies it by the following approximation:

$$\frac{(cos\mu_s + sunAngularRadius)^2}{4 * sunAngularRadius}$$

Precomputation of indirect irradiance assumes data in $\Delta\mathbb{S}$ be it single scattering if it is the first iteration of multiple scattering precomputation or one of the orders of multiple scattering. After altitude and sun-zenith angle cosine are determined from texel coordinates the integral in spherical coordinates for a hemisphere is numerically computed using trapezoidal rule. In case this is the first iteration, delta scattering should be fetched from two separate textures one for each scattering types with phase functions applied afterwards, since they are not applied for single scattering on precomputation as was suggested in Bruneton et al. algorithm (2). Otherwise the multiple scattering order of the previous iteration is fetched and applied directly, since phase function for the angle between main view-ray and sun direction is omitted only when accumulating delta in the scattering texture.

When indirect irradiance was precomputed in $\Delta\mathbb{E}$ it is rendered to the irradiance texture $\mathbb{E}$ with additive blending on.

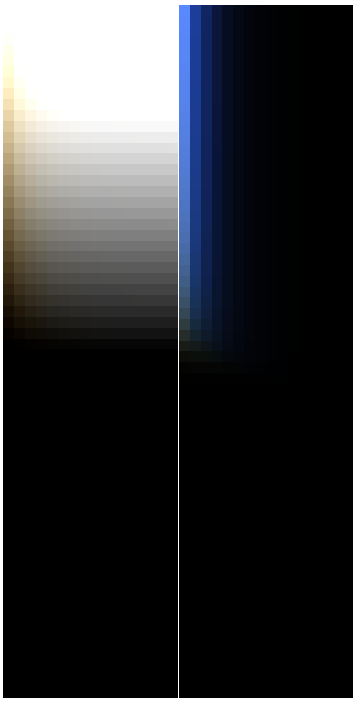

Figure 8.7 Precomputed direct irradiance and 4th order indirect irradiance

8.2.2.3 Scattering

The parametrization for altitude is modified as was described in **8.2.2.1 Transmittance.**

The view-zenith angle is also modified in a similar way, but since it has to account for rays intersecting both bottom and top atmosphere boundary and for discontinuity at the horizon, the parametrization is slightly different (25):

- $if\ r\mu < 0\ and\ \Delta > 0$

$$d = -r\mu - \sqrt{(r\mu)^2 - r^2 + R_g^2}$$

$$d_{min} = r - R_g$$

$$d_{max} = \rho$$
$$u_\mu = \frac{1}{2} - \frac{(d - d_{min})}{2(d_{max} - d_{min})}$$

- $otherwise$

$$d = -r\mu + \sqrt{(r\mu)^2 - r^2 + R_t^2}$$
$$d_{min} = R_t - r$$
$$d_{max} = \rho + H$$
$$u_\mu = \frac{1}{2} + \frac{(d - d_{min})}{2(d_{max} - d_{min})}$$

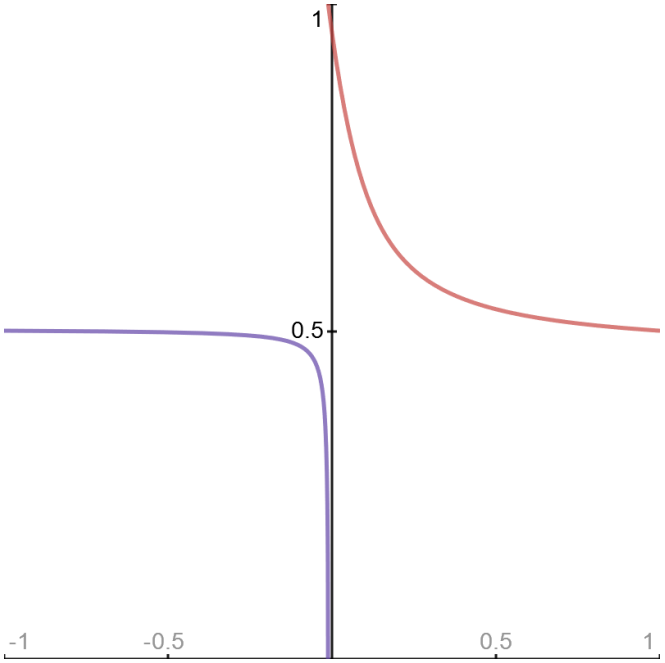

Figure 8.8 Parametrization of view-zenith angle cosine with Earth parameters $R_t = 6420\ km$ and $R_g = 6360\ km$ and altitude $r = 300\ m$

As seen from the diagram in **Figure 8.8** there is no gap between two functions on Y coordinate, the whole range $[0,1]$ is used for data. Since there are two different mapping functions in order their values would not mix and interpolate at the 0.5 edge, a special case of mapping boundary input values to the centre of the fragment should be considered:

$$normalizedCoordinates = \begin{cases} \dfrac{0.5n - inputCoordinate - 0.5}{0.5n - 1} & if\ r\mu < 0\ and\ \Delta > 0 \\ \dfrac{inputCoordinate - 0.5n - 0.5}{0.5n - 1} & otherwise \end{cases}$$

Parametrization for sun-zenith angle Bruneton et al. (2) provided is far from perfect since it is only applicable for Earth without any parameters to tweak. Much later Bruneton E. (25) came up with a significantly better solution that still uses ad hoc parameters but at least it provides flexibility in giving user the ability to control the boundary value for sun-zenith angle at which sun beyond horizon no longer contributes to lighting through scattering. With it being out of

the way, the artist is able to model lots of different atmospheres if he knows what he is doing, because the computation no longer relies specifically on Earth's parameters. This way more space is allocated for valuable data:

$$d_S = -R_g\mu_s + \sqrt{\left(R_g\mu_s\right)^2 + H^2}$$

d_S – Distance to the top atmosphere boundary along from the ground point along the sun ray.

$$d_{Smin} = R_t - R_g$$

d_{Smin} – Minimum possible distance to the top atmosphere boundary from the ground point.

$$d_{Smax} = H$$

d_{Smax} – Maximum possible distance to the top atmosphere boundary from the ground point.

$$a = \frac{d_S - d_{Smin}}{d_{Smax} - d_{Smin}}$$

a – Parametrization of the sun-zenith angle cosine in relation to the min/max possible distances to the top atmosphere boundary.

$$A = \frac{-2\mu_{Smin}R_g}{d_{Smax} - d_{Smin}}$$

A – Parametrized minimum sun-zenith angle cosine μ_{Smin} until cut off.

$$\bar{u}_{\mu_s} = \begin{cases} \dfrac{1 - \frac{a}{A}}{1 + a} & if\ a < A \\ 0 & otherwise \end{cases}$$

$\bar{u}_{\mu_s}$ – Sun-zenith angle cosine parametrization in relation to the cut off value. If sun-zenith angle is bigger than artist-set cut off, then data for this angle is precomputed in a a single row of texels.

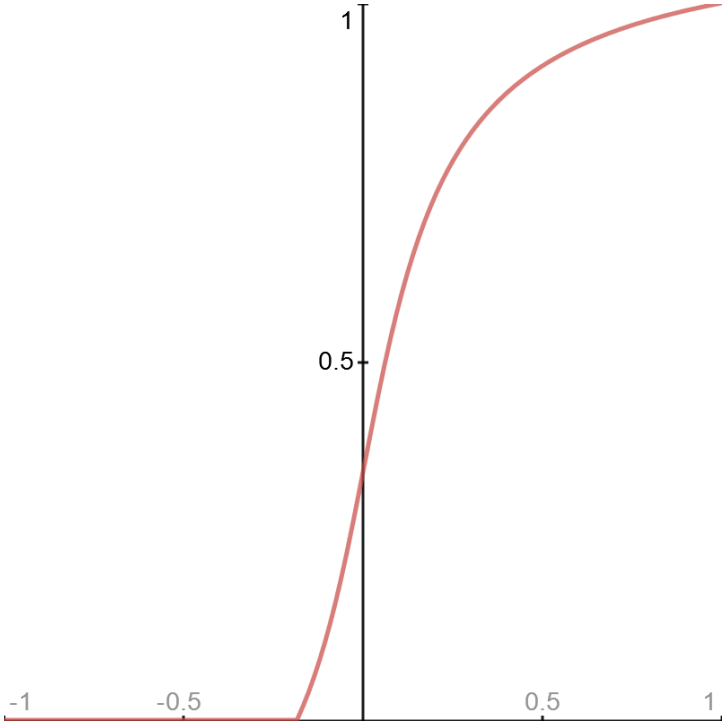

Figure 8.9 Parametrization of sun-zenith angle cosine with Earth parameters ($R_t = 6420\ km$, $R_g = 6360\ km$ and $\mu_{Smin} = -0.2$)

The last parameter, angle between sun and view direction ν is mapped linearly, the way it was suggested by Bruneton et al. (2).

Same as with irradiance, precomputation of scattering is also divided into two stages. First single scattering is precomputed without phase term in $\Delta\mathbb{S}$ and stored in $\mathbb{S}$, with Mie scattering stored separately (only red channel of single Mie scattering is stored in alpha channel of $\mathbb{S}$). Then multiple scattering is precomputed along with the phase term, but when it is added to the $\mathbb{S}$ the result of last precomputation is divided by Rayleigh phase term (both single Rayleigh and multiple Rayleigh and Mie scattering are getting divided by the same term that is initially lacking only in single Rayleigh scattering).

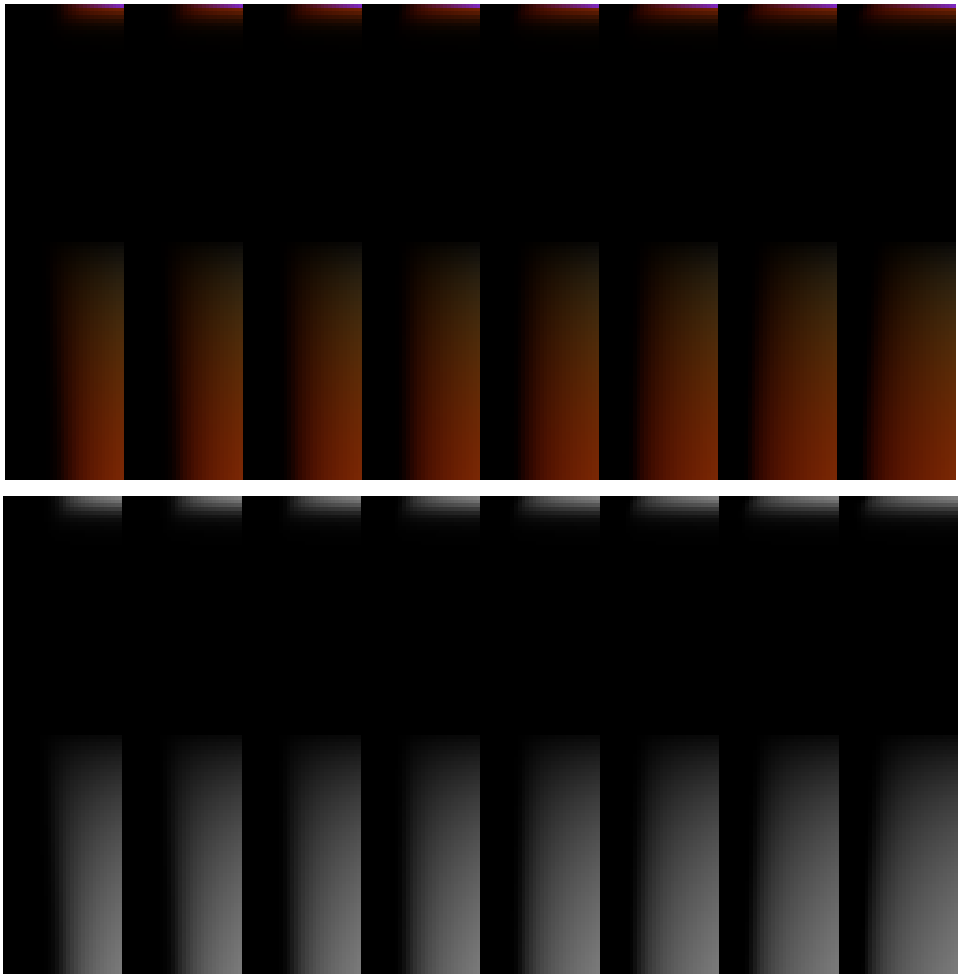

Figure 8.10 Single Mie scattering in RGB-channels of dedicated $\Delta\mathbb{S}$ (top) and in Alpha-channel of RGBA $\mathbb{S}$ (bottom)

Single scattering precomputation is done by taking samples along the ray defined by r and μ corresponding to the texel processed by fragment shader. At each sample transmittance to the sun is determined by fetching two values from $\mathbb{T}$. One is the transmittance to the sample point and the other to the top atmosphere boundary. The values at samples are integrated using trapezoidal rule the same way it was done for transmittance. Altitude and sun-zenith angle cosine at the sample point are determined the following way:

$$r_0 = \sqrt{r^2 + 2r\mu \cdot travelledDistance + travelledDistance^2}$$

$$\mu_{s_0} = \frac{r\mu_s + travelledDistance \cdot \nu}{r_0}$$

Precomputation of multiple scattering is done in two steps. First is a precomputation of $\Delta\mathbb{J}$ which is amount of inscattered light for particular order of scattering (number of bounces after single scattering) at any point in the atmosphere for any view ray and sun direction. Light bounces off the planet surface are included as well, that's why it is necessary to precompute direct irradiance in $\Delta\mathbb{E}$ even though it is not stored later in $\mathbb{E}$. The precomputation algorithm is similar to the indirect irradiance except this time integration in

spherical coordinates happens for a sphere and it is necessary to know the main direction vector to determine angle to each of the inscattered rays, so that it would be feasible to calculate phase term, which was omitted previously. Then, after indirect irradiance order is precomputed (since it depends on previous order $\Delta\mathbb{S}$), second step issues. This step accumulates all the precomputed values for sample points along the view ray. The process is almost the same as for single scattering precomputation, but instead of fetching transmittance values to the sun, the values of inscattered light at sample point from $\Delta\mathbb{J}$ is fetched. To fetch such value two additional parameters are necessary apart from r_0 and μ_{s_0} used for transmittance to the sun. View-zenith angle cosine μ_0 and view-sun angle cosine ν at the sample point. View-sun angle cosine is constant across all sample points for one ray, but view-zenith angle cosine still must be calculated:

$$\mu_0 = \frac{r\mu_s + travelledDistance}{r_0}$$

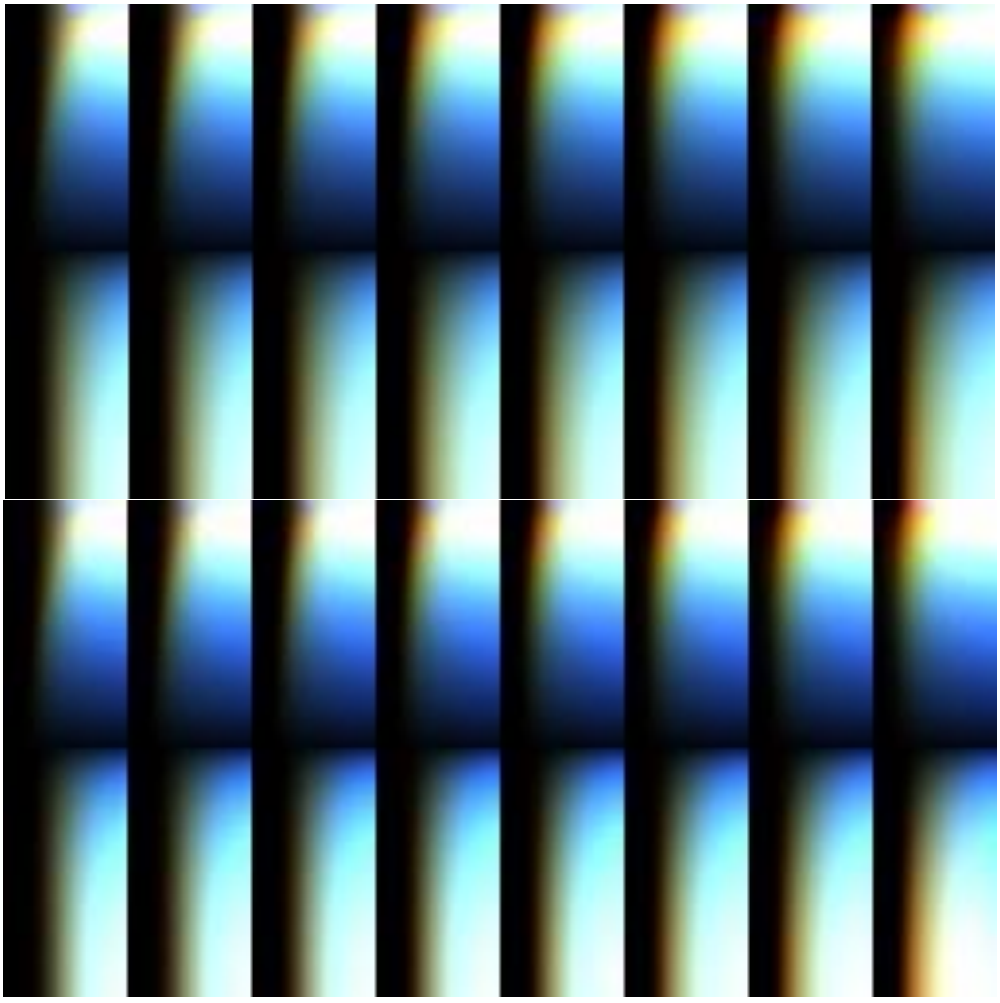

Figure 8.11 Single Rayleigh scattering (top) and multiple mixed scattering 4th order (bottom)

With each order of multiple scattering precomputed the data from $\Delta\mathbb{S}$ is accumulated to $\mathbb{S}$ with Rayleigh phase term for single scattering (view-sun angle) is omitted. However it is still included when calculating irradiance texture $\mathbb{E}$, which is tolerable since high precision is not

very important for correct lighting from ground reflection, hence phase function is not applied to the ground reflection at runtime.

8.2.2.4 Worley noise

After precomputations for atmospheric scattering are done it is time to move onto precomputations for clouds rendering. The first one is an inverted seamless 3D Worley noise.

Depending on frequency parameter, the normalized XYZ coordinates are scaled. Grid in this context is defined by vertices that are located at all integer coordinates. To determine to which cell voxel belongs it is sufficient to perform floor operation on voxel's coordinates, the result will be cell's coordinates in a grid. Likewise local coordinates of a voxel in a cell is a fractional part of its coordinates.

Since coordinates for each cell are unique, it is possible to get a unique point local to the specific cell using a hash function that inputs a vec3 and outputs a vec3 where each coordinate is in range [0, 1]. For this purpose the vec3 to vec3 integer based hash function by Inigo Quilez is used (27).

```
//Inigo Quilez 2017 Integer based (sinless) Hash
vec3 hash(vec3 cell)
{
    uvec3 uintCell = floatBitsToUint(cell);
    const uint k = 1103515245U;
    uintCell = ((uintCell>>8U)^uintCell.yzx)*k;
    uintCell = ((uintCell>>8U)^uintCell.yzx)*k;
    uintCell = ((uintCell>>8U)^uintCell.yzx)*k;

    return vec3(uintCell)*(1.0f/float(0xffffffffU));
}
```

Figure 8.12 Integer based hash by Inigo Quilez (27)

Based on that it is possible for any voxel in the grid to determine nine surrounding points, that are located each in its own corresponding cell with unique coordinates. One of these points will have the minimal Euclidean distance to that voxel among all points in the grid.

To make 3D Worley noise texture seamless each row, column and layer coordinates should loop if coordinates out of range $[0, \text{frequency}]$ are checked. This is manageable by the following formula:

$$y = \left(\left(1 + \left\lfloor \frac{||x||}{frequency} \right\rfloor\right) frequency + x\right) \bmod frequency$$

Which could be simplified, assuming the maximum distance from range to coordinates of a cell to check will be 1 (which is the case, since only eight neighbouring cells are checked apart from the one to which voxel belongs):

$$y = (frequency + x) \bmod frequency$$

In the end of all calculations for specific voxel it is inverted by subtracting result from 1.

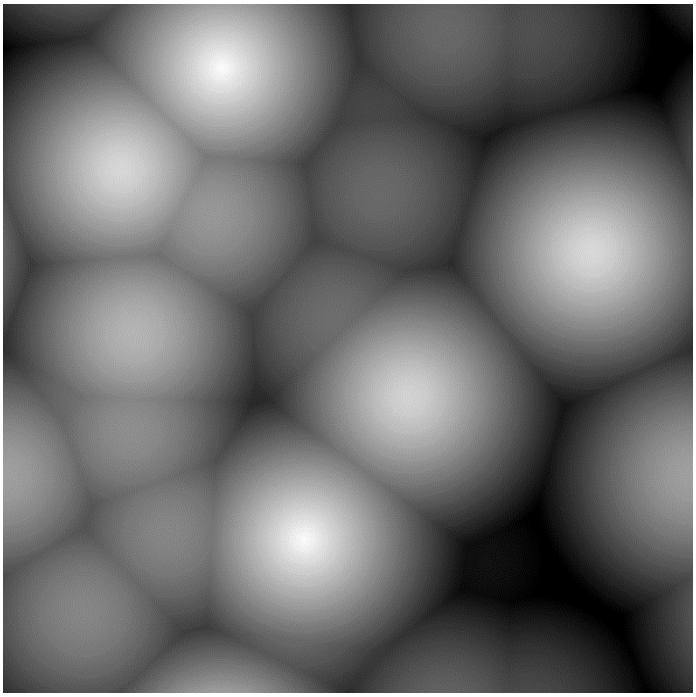

Figure 8.13 0th slice of 3D Worley noise of frequency 4

The Worley noise fBm is precomputed using the method suggested by Schneider A. (22):

$$\mathrm{fBm} = 0.625\mathrm{worley}(\mathrm{frequency}) + 0.25\mathrm{worley}(\mathrm{frequency} * 2) + 0.125\mathrm{worley}(\mathrm{frequency} * 4)$$

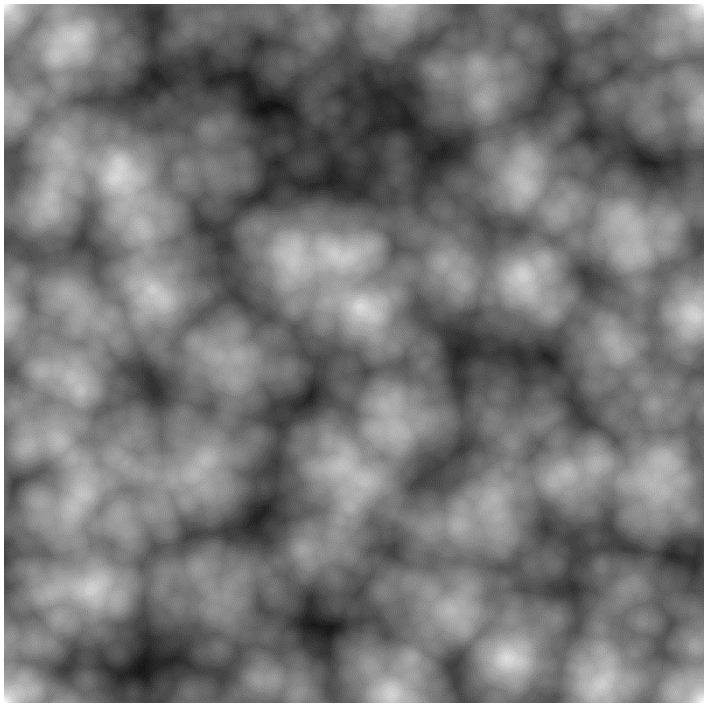

Figure 8.14 0th slice of 3D Worley noise fBm comprised of frequencies 8, 16 and 32

Five Worley noises of ascending power of two frequencies are precomputed which are combined into three fBms. Then these three fBms are combined in the same way.

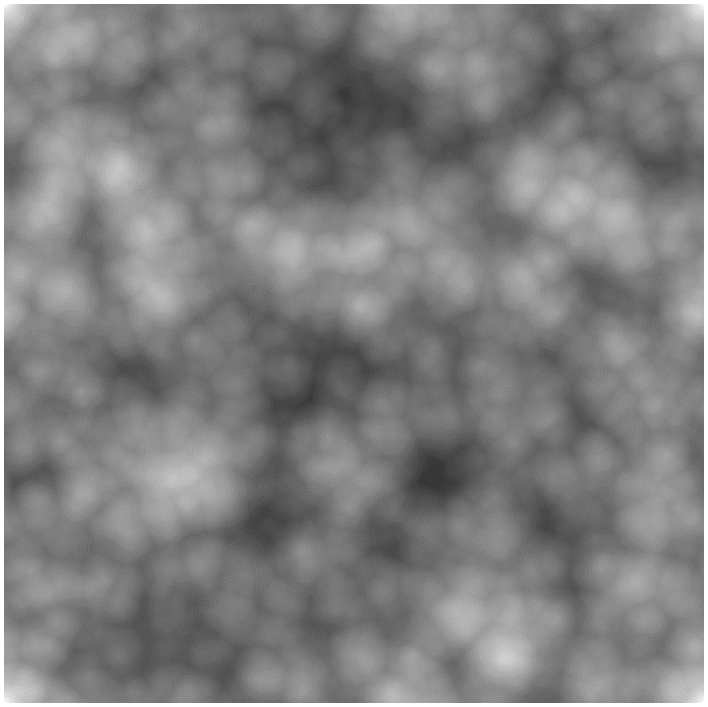

Figure 8.15 0^{th} slice of 3D Worley noise fBm comprised of 3 other Worley noise fBms of increasing frequencies

8.2.2.5 Perlin noise

Same idea as in **8.2.2.4 Worley noise** used to determine local voxel coordinates in respect to cell's coordinates to which it belongs. Local coordinates of each vertex of a cell are hardcoded. To get their global coordinates in a grid, these local coordinates must be added to cell's coordinates. This gives unique coordinates for each vertex. Just like before these coordinates are looped using aforementioned modulo formula. These coordinates are then put into hash function to get a unique gradient vector of the specific vertex. Gradient vector's coordinates must be mapped from $[0,1]$ range to $[-1,1]$, which is done simply by multiplying each coordinate by two and subtracting one. Then a vector from each vertex of the cell, to which voxel belongs, to voxel in question is calculated using local coordinates of both vertices and voxel. A dot product of each vector to vertex with gradient vector of the same origin is calculated. The resulting values for each vertex are then trilinearly interpolated using smootherstep interpolator by Ken Perlin (20) with voxel coordinates as input. Each smootherstep value is then used on the corresponding axis to interpolate. Perlin noise produces values in range $[-1,1]$ unlike Worley which range is close of $[0,1]$.

Figure 8.16 0th slice of 3D Perlin noise mapped to range [0,1]

FBm of Perlin noise is calculated using more classic approach. For each octave frequency gets increased by a power of two, while amplitude is being halved. Both noise and amplitude are independently accumulated after which noise is divided by accumulated amplitude value to normalize the result back to original range.

Figure 8.17 0th slice of 3D Perlin noise 7 octave fBm mapped to range [0,1]

8.2.2.6 Perlin-Worley noise

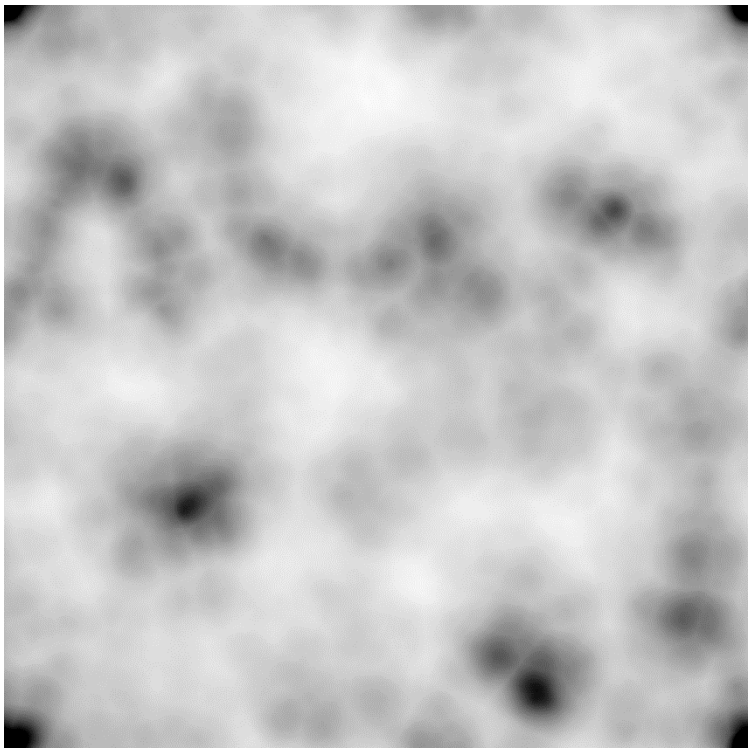

Figure 8.18 0th slice of a bad 3D Perlin-Worley noise mapped from range [-5.5,0.0] to range [0,1]

Doing exactly as suggested by Schneider A. in GPU Pro 7 book (22) by applying lower frequency Worley noise fBm as an old lower boundary value to remap Perlin noise didn't yield positive results. Not only the shape is the complete opposite of what is expected, the range of values is extreme as well. The output at 0th slice had a value evaluating to -280.

After some experimenting with inverting values and using them as different boundaries for old and new ranges, the conclusion was to use low frequency Worley noise fBm as a new lower boundary value. This way the results are much closer to examples from the book and there is no need to map to another range, values magically clip to [0,1]. Probably this is what was meant to be originally.

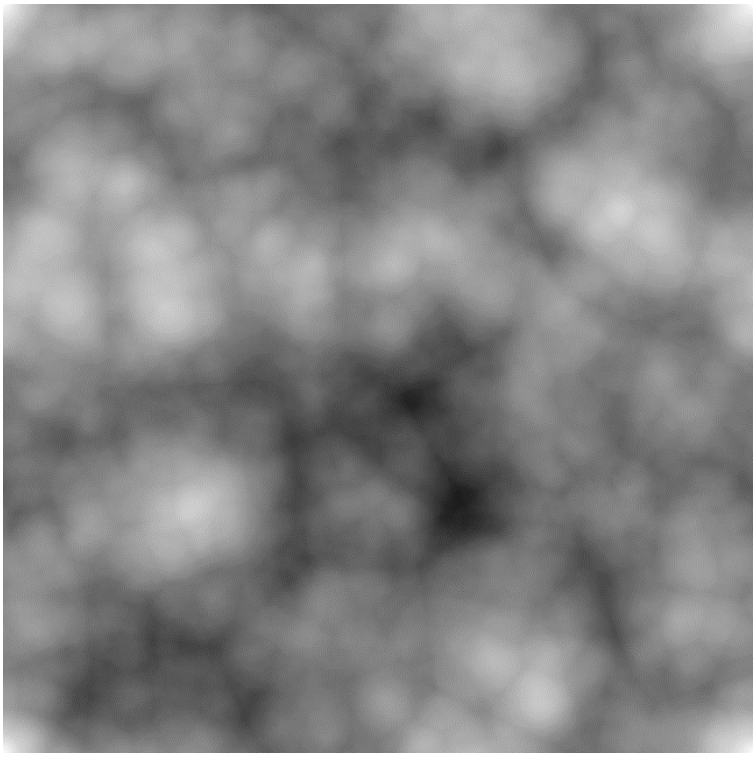

Figure 8.19 0th slice of a good 3D Perlin-Worley noise

So that no performance would be sacrificed, the shape of a cloud is dilated by remapping Perlin-Worley noise with inverse combined Worley noise fBm as an old lower boundary value, the way Schneider A. suggested to do it in realtime (22).

8.2.3 Rendering

The actual rendering of effects happens in the renderAtmosphere method in the Renderer class. First it renders to offscreen FBO, which allows usage of non-standard textures types for debugging purposes, and then renders the offscreen rendered texture to the screen. The most of the workload is done in planetShader, which renders basically everything except for user interface which is Qt's responsibility.

planetShader makes use of a full screen triangle vertex shader same as shaders in precomputation stage. A ray, which is rotated with viewMatrix is sent through each vertex of a triangle and then gets interpolated for each fragment. The fragment shader is what makes use of all the preparations done until now.

The first step it does is if the viewer is outside the atmosphere, it determines whether he is looking into it. If so, the viewer gets moved to the atmosphere boundary along the ray. Everything related to his position and zenith angles then gets reassigned or recalculated.

Next, after last recalculations, it is safe to assume that the viewer is either inside the atmosphere or on its boundary if his view ray goes through the atmosphere. For atmosphere rendering there are two slightly different cases:

- The viewer is looking at the sky.
- The viewer is looking at the planet's surface.

For both cases inscattering should be evaluated. First, inscattering is fetched from the texture with manual linear interpolation on 4th coordinate. The fetched inscattering is a mixed inscattering. In the first three channels a single and multiple Rayleigh scattering and multiple Mie scattering are stored without phase term for single Rayleigh scattering. In the last, alpha channel the red channel of single Mie scattering without corresponding phase term is located. The only way to get all other channels of single Mie scattering is to extrapolate them from the given data the way it was suggested by Bruneton et al. (2) and discussed in the end of **5.2.2.3 Precomputation**. After all channels of both single Mie scattering and mixed scattering are available it is time to multiply them both by respective phase functions. This way of deferring phase function multiplication to real time for single scattering is working only due to assumption that all sun rays are parallel and the sun is considered *directed light.*

Once inscattering is dealt with the only part that is left is to do for the first case is to add direct light contribution from the sun, which is transmittance to the top atmosphere boundary along the viewing ray times direct sun light in that direction. To determine the direct sun light the view-sun angle should be lesser than the angular radius of the sun. Then it is possible to smoothen out the sun disk by making the solar radiance a bit smaller closer to the edges:

$$L_{sun} = \frac{\nu - \cos\ sunAngularRadius}{1 - \cos\ sunAngularRadius} solarRadiance$$

In the second case however amount of inscattering should be dependent on whether view ray hits shadow volumes, but since in this project the planet is ideally spherical, there is no terrain to force shadow calculations, hence inscattering is uniform and is independent of shadows. But still since the view ray hit a surface, the reflection should be computed in addition to the contribution from inscattering. The transmittance to the point of intersection is multiplied by Lambertian BRDF $\frac{\alpha}{\pi}$ and ground irradiance which is a sum of both direct and indirect irradiance. Direct irradiance is calculated in real time by multiplying transmittance from intersection point to the top atmosphere boundary in the direction of the sun with solar irradiance value and cosine of sun-zenith angle at surface intersection point which defines diffuse lighting. It is also possible to approximate for ambient occlusion by mapping cosine of non-horizontal surface normal-zenith angle to $[0,1]$ and multiplying it with the result. As for indirect irradiance it is much simpler since it is already precomputed and all that is left is to fetch correct values from the texture for altitude at surface level and sun-zenith angle cosine, which are the same parameters used for computing direct irradiance.

Right after rendering the atmosphere the parameter that controls opacity of the clouds is checked to determine whether they should be rendered at all. Then a check on intersection of the cloud layer is issued which is a bit more convoluted than atmosphere intersection test,

since it is possible to be inside, outside and inbetween both boundaries. But in the end it is just amount of sphere intersection checks since it doesn't really matter which exactly boundary is intersected unlike it was with the atmosphere where depending on whether it was top or bottom boundary different algorithms issued. What is necessary to determine is the distance inside the layer, since amount of accumulated density depends on that.

Once distance of ray inside cloud layer was determined the transmittance inside clouds is calculated by numerically solving integral the similar way transmittance for atmospheric scattering was precomputed. The only difference is that due to time constraints it was unfeasible to figure out light interaction calculations, hence the solution is far from simulation of a physical model and uses generated noise textures to accumulate cloud density directly and then approximate transmittance using natural exponential function. The higher the density the lower is the transmittance.

There are many hard coded parameters which are not tweakable from the user interface since getting plausible results is extremely difficult and is a matter of trial and error which could last from several hours to days if the user is not familiar with the software:

- The gradient modifier modifies the accumulated values depending on the altitude as clouds on higher altitudes are less dense.
- Cloud layer altitude is fixed above planet's surface.
- Steps is the amount of steps done during integration. The step size is determined depending on that.
- Density multiplier multiplies the final accumulated cloud density, but also affects early exit from the integral when large enough density was already accumulated.
- Cut off density is the amount of density is necessary to accumulate until it is safe to stop integration without loss in quality.
- Cloud frequency increases 3D texture frequency out of which the voxelized cloud layer is made through which ray passes (sample world position is used directly in the noise texture sampler as was suggested by Ken Perlin (28), otherwise horrible mapping artifacts are present due to the fact that a very large sphere requires a large amount of detail, which is impossible without texture repeat in case of 3D textures, which makes artifacts extremely obvious).
- Detail frequency controls texture frequency when sampling noise for cloud edge eroding for additional detail.
- The amount of clouds and their density is also highly dependent on the final remap done in noise precomputation to normalize values to a range $[0,1]$.

Once the transmittance is approximated the rendering output is calculated using the following formula:

$$atmosphere \cdot cloudTransmittance + (1 - cloudTransmittance) \cdot inscatter \cdot cloudOpacity$$

The render is essentially a blending between atmosphere render and highlighted zones of low transmittance due to clouds with fake light scattering inside clouds influenced by sky and atmosphere colour.

The clouds are also controlled by cameraOpacity parameter in the main function which makes them fade away with altitude. Volumetric cloud geometry is not seen from space anyway due to large distances. This makes solution using 2D cloud textures for a sphere a much better option when rendering clouds from space.

Finally, if a view ray does not intersect atmosphere at all, then the only light that gets rendered is direct sun light with no attenuation.

After all light calculations were finished and radiance is accumulated the last necessary step is tone mapping and gamma correction. In this implementation this is done in a combined function that targets 2.2 gamma with exposure and white point for each individual colour with controls from user interface.

8.2.4 Default parameters

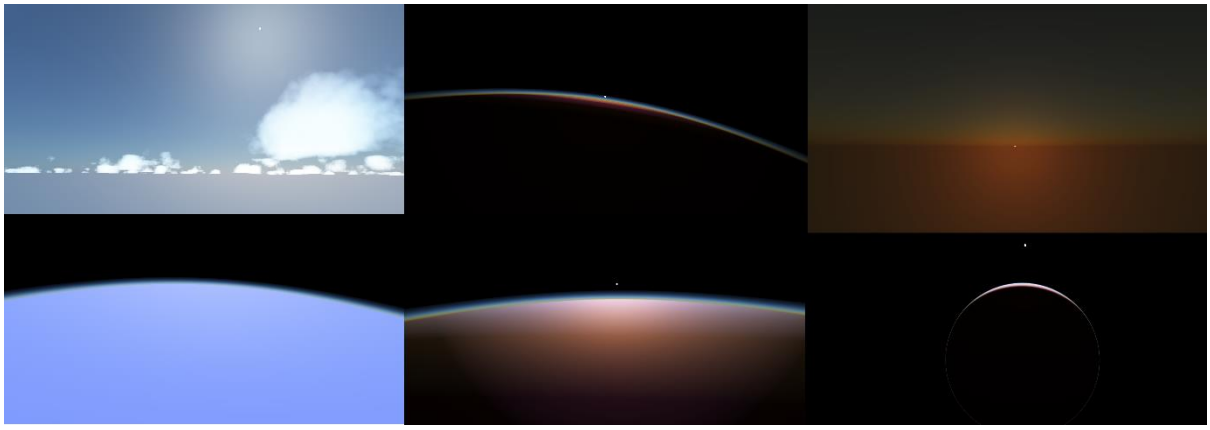

Figure 8.20 Screenshots done in software launched with default parameters

Parameters are mostly based on the ones provided by Bruneton et al. (2) with some changes due to personal preferences.

Table 8.2 Physical parameters

R_g	6360 km
R_t	6420 km
Solar Irradiance	1.5
μ_{Smin}	-0.2
Sun Angular Radius	0.0046251225

β_R	$5.8 * 10^{-6}, 13.5 * 10^{-6}, 33.1 * 10^{-6}$
β_M	$2.21 * 10^{-5}$
H_R	8 km
H_M	1.2 km
g	0.73

Table 8.3 Texture and rendering parameters

Transmittance Texture Width (r)	64
Transmittance Texture Height (μ)	256
Transmittance Samples	500
Irradiance Texture Width (r)	16
Irradiance Texture Height (μ_s)	64
Irradiance Spherical Samples	32
Albedo (α)	0.0,0.0,0.04
Order Count	4
Exposure	10
White point	$2.8, 2.175, 1.875$

Scattering Texture Layers (r)	32
Scattering Texture Height (μ)	128
Scattering Texture Width 1* (μ_s)	32
Scattering Texture Width 2* (ν)	8
Scattering Spherical Samples	16
Scattering Samples	50
Cloud Noise Texture Width	128
Cloud Noise Texture Height	128
Cloud Noise Texture Depth	128
Cloud Opacity	1

Chapter 9
Results and Evaluation

9.1 Benchmarks

9.1.2 Precomputation

Table 9.1 Benchmark results

Default settings	113 ms
Size of scattering texture x81	6928 ms
Size of transmittance texture x10000	1113 ms
Size of irradiance texture x10000	970 ms

The table clearly shows that the most performance heavy precomputation is done for scattering, which is expected since scattering texture precomputation involves two steps with three integrals, two of which are nested one into another to perform full spherical integration.

Table 9.2 Memory trade off

T	96KB
E, Delta_E	6KB x2
S	8MB
Delta_J, Delta_SR, Delta_SM	6MB x3
Noise	4MB
Overall	20MB 118KB

9.1.3 Rendering

The biggest performance hit is due to the clouds rendering. Since there was no particular effort made in order to optimize it, it performs constant sampling of data 200 steps per ray with two texture samplings per step. Disabling clouds speeds up rendering from 3ms to 1ms per frame. Performance is completely independent of size of precomputed textures for atmospheric scattering. Doesn’t matter whether 4k or 16k resolution textures are used performance stays roughly the same at 1ms per frame with clouds disabled., However using 1280x1280x128 noise texture for clouds rendering instead of default 128x128x128 looses

12ms of render time per frame and when moving viewer position and orientation frames per second constantly drop to values lower than 60, which is unacceptable.

Generally speaking the best method to increase performance would be to lower the resolution or frequency of noise texture and decrease amount of sampling when rendering clouds. Unfortunately, this is extremely hard to do without loosing visual fidelity, which on its own is far from high for clouds in this implementation. Considering that, it would be much easier to disable clouds altogether in order to increase performance. Otherwise, instead of spending time tweaking, it would be much more useful to come up with some other more optimized way of rendering them for example by rendering them at lower resolution and then applying temporal reprojection (22).

9.2 Personal reflection

Development of this project was unprecedently turbulent. Starting off initially as a group of two, picking up the task officially meant for at least three, the first attempt didn't last long and the whole project required redefining halfway through. And this was only one (probably not even the most significant) of many obstacles that had to be surmounted over the course.

From the very beginning when me and my partner were unable to come up with a cohesive plan, both our supervisor and assessor pointed out how important it is to have a strong communication in order to stay informed and how everything will fall into place once every member of the group will see the full picture.

Later on this proved to be a very important lesson. Other issues aside, when global pandemic started, staying in contact helped the most, since at times when nobody is sure about what comes next it is easy to detract from the chosen path, yet by supporting order and making sure that everyone reaches understanding and sets realistic expectations, there was no hindrance in sticking to the plan which was proposed before actual problems arose.

Initially plan for individual project was much richer, with many other natural phenomena included, but once first advancements in research were made it became more apparent that atmospheric scattering on its own is a very dense topic reading about which I could spend subsequent months and still not grasp everything completely. After some time, a final implementation of the modern approach to solve this problem was done. The performance was impressive, nothing like I've seen on a CPU before. Still there was some time left.

Finally it was decided that volumetric clouds would make a fine addition to the atmospheric scattering project. During this remaining time I had to learn a bit more in depth about volumetric rendering and try my best to implement it producing something resembling clouds in the process. Over the course of studying I have learned how to precompute different types of 3D noises, but what came as an unexpected challenge is the noise mapping on a sphere.

It appeared that wrapping a whole planet with a base layer of a 3D noise when raymarched generated nothing that would feel like clouds in the sky, more like some generic repeating pattern full of artifacts either squishing the 3D texture or not even using half of the generated data.

Luckily, some of these issues were mitigated, but time constraints didn't let me research the topic more thoroughly and come up with a more elegant solution.

Besides that during the suspension and outbreak I was able to dramatically improve my skills in computer graphics. Before I stopped working on the project due to illness I only had a very slow poor CPU implementation of a single scattering that was calculated on the fly for extremely low resolution. Once I could spend more time home I gradually made my mind up and started learning more about modern OpenGL which I never had experience with before. Ultimately not only it was an absolutely right decision to make and I definitely did not waste my time on some irrelevant activity, I've got to the point where I can not imagine how I would go about implementing this project without any of that knowledge I have acquired.

I cannot be grateful enough to see that this undertaking ended up being a success. To me it is a great accomplishment to be able to keep up and achieve even when it seems as if, despite being unlikely, negative events never stop popping up.and making you question yourself whether at such times it is worth it at all.

Chapter 10
Conclusion

10.1 Summary

This project explored several different themes starting from game engine development and ending up with photorealistic rendering of natural phenomena. The two phenomena this project focused on was atmospheric scattering and clouds. While atmospheric scattering works incredibly well, there is still much to improve for volumetric clouds.

10.2 Further Work

There are many aspects upon which the current iteration of the software could be improved. The most logical one is to work on features that were not implemented fully or were not integrated well in the existing application. There is still much to improve in this project's volumetric rendering of clouds because in its current state it is far from the modern solutions and is more akin to experiments of someone who never did this before, which is actually the case.

Complex physically based light behaviour and its interaction with small particles inside clouds could be researched further to implement an actual simulation of a physical model instead of something faking it. Along with that, atmospheric scattering light contribution could influence lighting of clouds and vice versa. This would most likely produce far more realistic picture of both clouds and atmosphere.

Another topic to look into is texture parametrization as it was done by Bruneton et al. (2). For example, it should be possible to improve sun-zenith angle parametrization so that it would not rely on some parameter which is not easily derivable, but is rather guessed by artists. Almost half of the irradiance texture is not used at all (look at **Figure 8.7**) due to scattering not contributing light to the surface of the planet when sun is far beyond the horizon. A good idea would be to make parametrization dependant on the sun and scattering parameters, so that more data would be allocated to currently unused space.

List of References

1. Nishita, T., Sirai, T., Tadamura, K. and Nakamae, E. Display of the earth taking into account atmospheric scattering. In: *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques.* New York, 1993, pp. 175 – 182.
2. Bruneton, E. and Neyret, F. Precomputed atmospheric scattering. In: *EGSR '08: Proceedings of the Nineteenth Eurographics conference on Rendering.* Goslar, 2008, pp. 1079 – 1086.
3. Chandrasekhar, S. *Radiative Transfer.* New York: Dover Publications, 1950.
4. Riley, K., Ebert, D. S., Kraus, M., Tessendorf, J. and Hansen, C. D. Efficient rendering of atmospheric phenomena. In: *EGSR '04: Proceedings of the Fifteenth Eurographics conference on Rendering Techniques.* Goslar, 2004, pp. 375 – 386.
5. Bruneton, E. A Qualitative and Quantitative Evaluation of 8 Clear Sky Models. In: *IEEE Transactions on Visualization and Computer Graphics.* **23**(12), pp. 2641 – 2655.
6. Microsoft Flight Simulator. Microsoft Windows [Game]. Xbox Game Studios: Redmond, Washington, 2020.
7. Battlefield 4. Microsoft Windows [Game]. Electronic Arts: Redwood City, California, 2013.
8. Uncharted 4: A Thief's End. PlayStation 4 [Game]. Sony Computer Entertainment: Tokyo, San Mateo, California, 2016.
9. Red Dead Redemption 2. PC [Game]. Rockstar Games: New York, 2019.
10. Horizon Zero Dawn. PlayStation 4 [Game]. Sony Computer Entertainment: Tokyo, San Mateo, 2017.
11. Blinn, J.F. Light reflection functions for simulation of clouds and dusty surfaces. *SIGGRAPH Comput. Graph.* 1982, **16**(3), pp. 21 – 29
12. Schafhitzel, T., Falk, M. and Ertl, T. Real-time rendering of planets with atmospheres. *Journal of WSCG.* 2007, **15**, pp. 91 – 98
13. O'Neil, S. Accurate Atmospheric Scattering. In: Pharr, M. ed. *Gpu Gems 2*. Massachusetts: Addison-Wesley Professional, 2005, pp. 253 – 268
14. Stamnes, K., Thomas, G., and Stamnes, J. *Radiative Transfer in the Atmosphere and Ocean*. Cambridge: Cambridge University Press, 2017.
15. Cornette, W. M., and Shanks, J. G. Physically reasonable analytic expression for the single-scattering phase function. *Applied optics.* 1992 , **31**(16), pp. 3152 – 3160
16. Unity Technologies. Unity (2019.3.13). [Software]. 2020. [Accessed 30 May 2020].
17. Gregory, J. *Game engine architecture.* 3rd ed. Natick: AK Peters, 2014.
18. Epic Games. Unreal Engine (4.25). [Software]. 2020. [Accessed 30 May 2020].
19. Perlin, K. An image synthesizer. In: *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, 22-26 July 1985, San Francisco.* New York: Association for Computing Machinery, 1985, pp. 287 – 296

20. Perlin, K. Improving noise. In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, 21-26 July 2002, San Antonio.* New York: Association for Computing Machinery, 2002, pp. 681 – 682
21. Worley, S. A cellular texture basis function. In: *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, 4-9 August 1996, New Orleans.* New York: Association for Computing Machinery, 1996, pp. 291 – 294
22. Schneider, A. Real-Time Volumetric Cloudscapes. In: Engel, W. ed. *Gpu Pro 7: Advanced Rendering Techniques*. Natick: AK Peters, 2016, pp. 97 – 127
23. Mandelbrot, B.B. and Ness, J.W. Fractional Brownian motions, fractional noises and applications. *Society for Industrial and Applied Mathematics.* 1968, **10**(4), pp. 422 – 437
24. Lottes, T. *FXAA.* [Online]. Santa Clara: Nvidia, 2011. [Accessed 30 May 2020]. Available from: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf
25. Bruneton, E. *Precomputed atmospheric scattering: a New Implementation.* [Online]. 2017. [Accessed 30 May 2020]. Available from: https://ebruneton.github.io/precomputed_atmospheric_scattering/
26. Martinez, I. *Radiative View factors.* [Online]. 2020. [Accessed 30 May 2020]. Available from: http://webserver.dmt.upm.es/%7Eisidoro/tc3/Radiation%20View%20factors.pdf
27. Quilez, I. *Integer Hash – II.* [Online]. 2017. [Accessed 30 May 2020]. Available from: https://www.shadertoy.com/view/XlXcW4
28. Perlin, K. *Making Noise.* [Online]. GDCHardcore, 9 December 1999. [Accessed 30 May 2020]. Available from: https://web.archive.org/web/20071008162042/http://www.noisemachine.com/talk1/index.html

Appendix A
External Materials

3rd-party libraries:

- GLU 5.13.0 – Core, Widgets, OpenGL, GUI
  https://www.opengl.org/resources/libraries/
- Qt https://www.qt.io/

Appendix B
Ethical Issues Addressed

This project did not deal with any ethical issues.