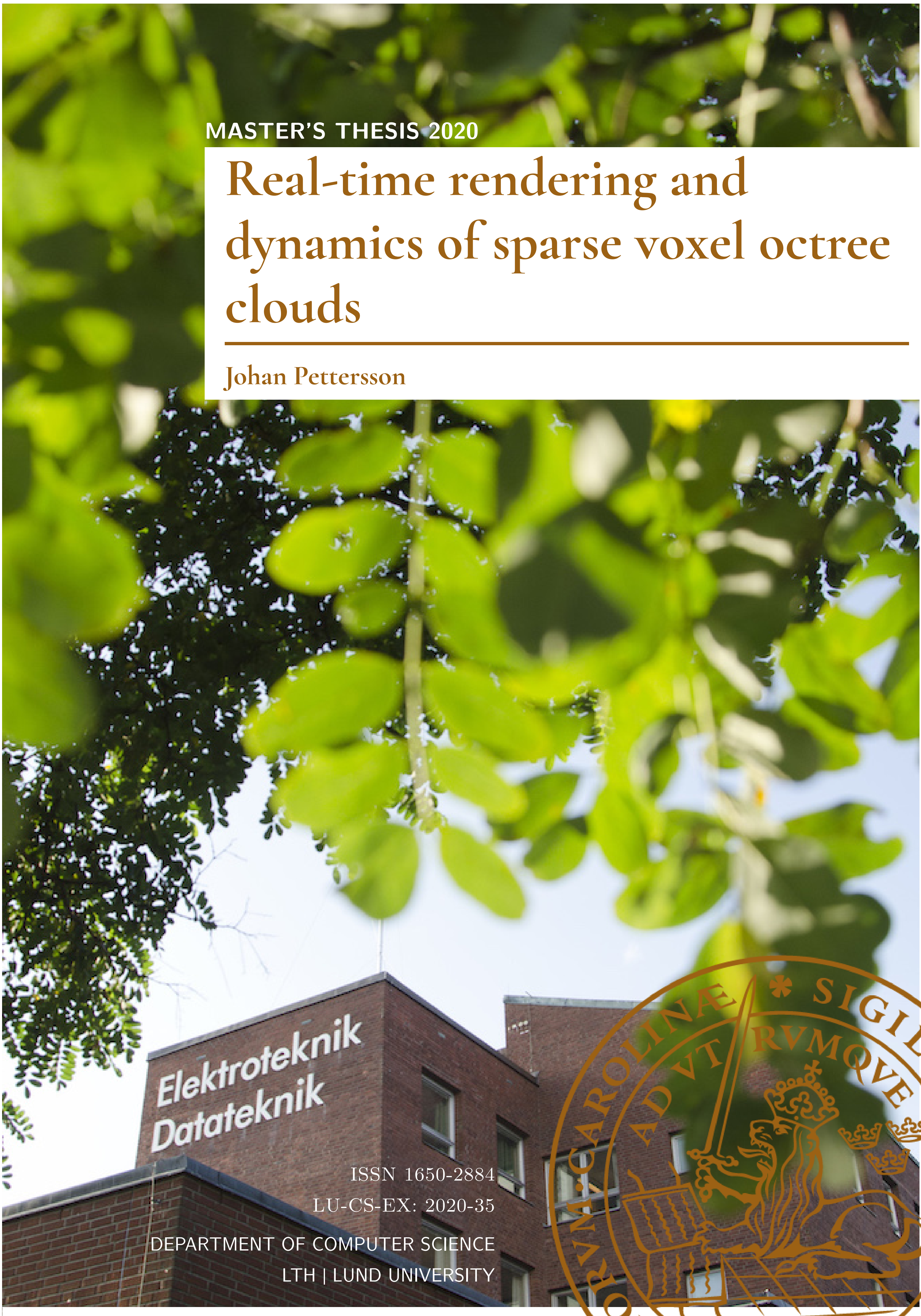
MASTER'S THESIS 2020

Real-time rendering and dynamics of sparse voxel octree clouds

Johan Pettersson



ISSN 1650-2884
LU-CS-EX: 2020-35

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-35

Real-time rendering and dynamics of sparse voxel octree clouds

Johan Pettersson

Real-time rendering and dynamics of sparse voxel octree clouds

Johan Pettersson
dat14jpe@student.lu.se

June 16, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Pierre Moreau, pierre.moreau@cs.lth.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

State-of-the-art real-time cloud rendering in games has transitioned from 2D skyboxes to ray marched 3D volumes more closely approximating the volumetric nature of reality. However, most solutions repeat small volumes of details to save memory and show only a few hardcoded cloud layers, which cannot capture the breadth and depth of complexities in the real atmosphere.

This thesis develops and presents an atmosphere renderer capable of displaying views from any altitude within or outside an Earth-scale atmosphere, seamlessly transitioning between them, with adaptive level of detail changing the allocation of memory and computational resources depending on the current camera location. By separating cloud creation from cloud rendering, this method in principle enables rendering of cloud data from any source.

Ultimately the presented implementation renders on average 30 frames per second with adequate quality settings on a high end GPU; however, there may be further optimisation potential.

Keywords: level of detail, voxels, rendering, computer graphics, image rendering, real-time rendering

Acknowledgements

I would like to express deep gratitude to my parents Catarina and Nils, for letting me keep my head aloft in the clouds. Furthermore, I thank my supervisor Pierre Moreau for putting up with the vagaries of a voxel enthusiast, and for his stewardship of a plastic box devoid of cookies during the trying times of pandemic. Likewise, I give kudos to Michael Doggett for fair remarking and marking. Additionally, I profess my unwavering gratefulness to Simon Bengtsson, Oscar Sigurdsson, and Christopher Tvede-Möller for proofreading a provably proofless thesis. Moreover, I recognise my indebtedness to Markus Åkesson, Filip Andersson, Sigrid Berglund, Kristoffer Bergman, Daniel Bjarnestig, Filip Fläderblom, Michael Hansen, Dennis Jin, David Malmström, Alexander Olsson, Markus Oskarsson, Patrik Persson, Eric Sporre, and Andy Tang for benchmarking the program on various GPUs. Finally, I proclaim heartfelt thankfulness to Sångfågel the cat for unending impromptu jam sessions in rainfall and sunshine alike.

Post-ultimately, thank *you* for reading this far.

Contents

Chapter 1

Introduction

The sky comprises a significant portion of many outdoor scenes, literally setting the atmosphere and playing a major part in guiding mood and tone. While a clear blue day has a certain appeal, the dynamic conditions of weather will frequently give rise to a wide array of conspicuous visible formations of aerosolised water, commonly known as clouds. There are many types of clouds, from fluffy cumulus to blanketing stratus and wispy cirrus, and the occasional towering cumulonimbus, as well as various subtypes thereof (see Section 2.1.1 on page 13); their varied configurations interact with sunlight to enact complex displays of light and shadow.

Rendering the atmosphere is thus a recurring sub-objective in computer graphics. Different solutions have been devised over the years, with the most recent techniques seemingly converging on variants of ray marching a cloud density volume to compute approximate in-scattering and transmittance from a given direction to the viewing camera. However, most real-time methods limit views to positions near the ground, and few enable continuous movement from vantage points overlooking an entire planet, through fully volumetric cloud layers, and to arbitrary levels.

In this thesis I present a newly developed atmosphere renderer intended to allow views over Earth-scale planetary distances and (see Figure 1.2 on page 9) from any perspective either within (see Figure 1.1 on the following page) or outside (see Figure 1.2 on page 9) the atmosphere. The first-order light scattering caused by smaller molecules is computed according to a density distribution dependent only on height above sea level, as in earlier works; however, the presented solution also accounts for first-order light scattering due to the spatially-variable density distribution of water droplets suspended in the air (i.e., clouds and fog) using an adaptive spatial partitioning structure.

1.1 Goals

The atmosphere renderer developed as part of this thesis work was named *Mulen* (Swedish for "overcast" or "cloudy", as in "en mulen dag" i.e. "a cloudy day"). These were the goals:

- Real-time performance, defined as no fewer than 30 frames per second (FPS) when running on an NVIDIA GTX 1080 Ti graphics card and rendering to a resolution of 2560×1440 pixels.
- Capacity to handle a full realistically sized planetary atmosphere (planet radius $6371\,\mathrm{km}$ and atmosphere shell approximately $100\,\mathrm{km}$ thick around that, like Earth), with few or no abrupt transitions (i.e. seamless movement around the planet, and through the clouds).
- Dynamic cloud movements and dynamic lighting, precluding precomputed (baked) lighting solutions.

Figure 1.1: Parts of a cumulus cloud photographed in late afternoon.

1.2 Contributions

Mulen was intended to investigate the feasibility of creating and rendering planet-encompassing cloudscapes in real-time using a variable-resolution voxel structure. If successful, such an approach could be used to display any cloud formation permissible in 3D space, including any procedurally generated ones as well as ones scanned from reality, and in particular lead the way to player-modifiable clouds of arbitrary complexity (insofar as is permitted by hardware memory and computational resources).

A list of specific contributions:

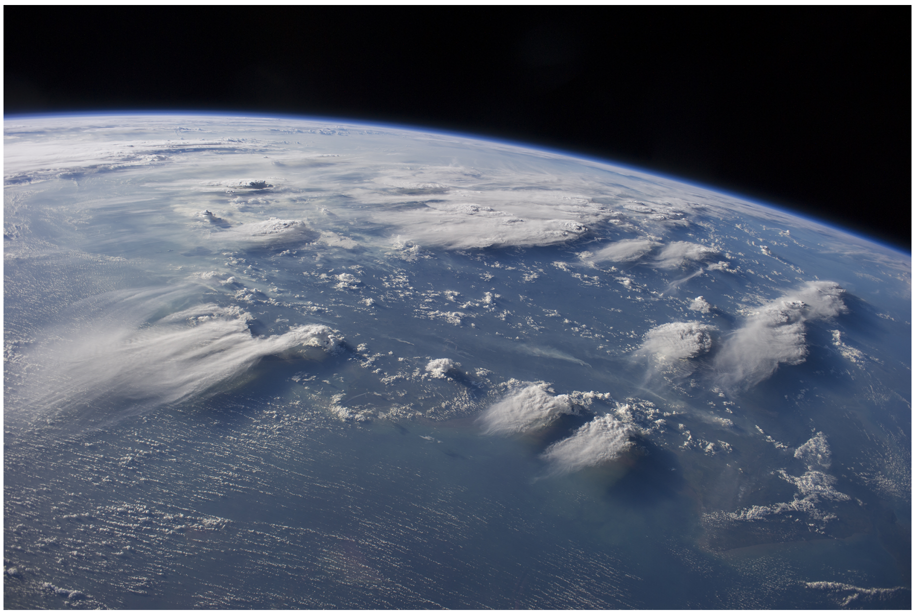


Figure 1.2: Thunderheads near Borneo, Indonesia featured in an image photographed by an Expedition 40 crew member on the International Space Station (NASA photo ID ISS040-E-88891). Image courtesy of the Earth Science and Remote Sensing Unit, NASA Johnson Space Center [5].

- Application of a sparse voxel octree to real-time atmosphere rendering, allowing for separation of cloud generation from cloud rendering.
- Real-time construction and adaptation of the octree on another thread depending on current and possibly changing camera location.
- A method of animating the sparse voxel octree by interpolating between two states.
- Render performance independent from computation of cloud distribution due to separation of cloud generation from cloud rendering.

1.3 Report structure

After this introductory chapter the theoretical foundations of this work are introduced, divided into concepts from physics and concepts from computer graphics. These are followed by a roughly chronological overview of related works that inspired and helped guide the approach and aims of this thesis. Subsequently the algorithmic structure of Mulen is explained. Then some technical specifics of its implementation (such as crucial optimisations and design tradeoffs) are detailed. The results are presented – both those which are directly visual, in the form of images and linked videos, as well as performance measurements. This leads into a discussion of the advantages and disadvantages found in the explored method, with suggestions of future work and some improvements which might deserve looking into. Finally, a brief conclusion.

Chapter 2
Theory

This chapter summarises the high-level physical foundations of atmospheric lighting, as well as some essential computer graphics concepts employed in this thesis. The two types of light scattering (Rayleigh and Mie), as well as light absorption due to ozone, are introduced. An overview of the process behind cloud formation is presented.

The modelling of colours is covered, as well as tone mapping from high dynamic range to low dynamic range. The octree spatial partitioning structure is motivated in the context of this thesis and described.

2.1 Atmosphere physics

An atmosphere is a shell of gaseous matter surrounding a celestial body such as a planet or a natural satellite (i.e. a moon). In this work, three types of atmosphere constituents are considered, as they are the most relevant for imitating the visual characteristics of planet Earth's atmosphere (see Figure 2.1 on the following page):

- Molecules whose sizes are significantly smaller than the wavelengths of visible light, such as oxygen and nitrogen.
- Water droplets, whose sizes are similar to or exceed the wavelengths of visible light.
- Ozone (which is technically included among the molecules, but warrants separate handling due to having a particular visual effect and a separate density distribution).

The densities of both smaller molecules and water droplets in clear weather are modelled as exponential distributions over height above ground level:

$$\rho(h) = e^{-h/H} , \tag{2.1}$$

where e is Euler's number and H is the distribution-dependent constant *scale height* (8 km for smaller molecules and 1.2 km for water droplets, which means the latter are relatively scarcer in the upper atmosphere).

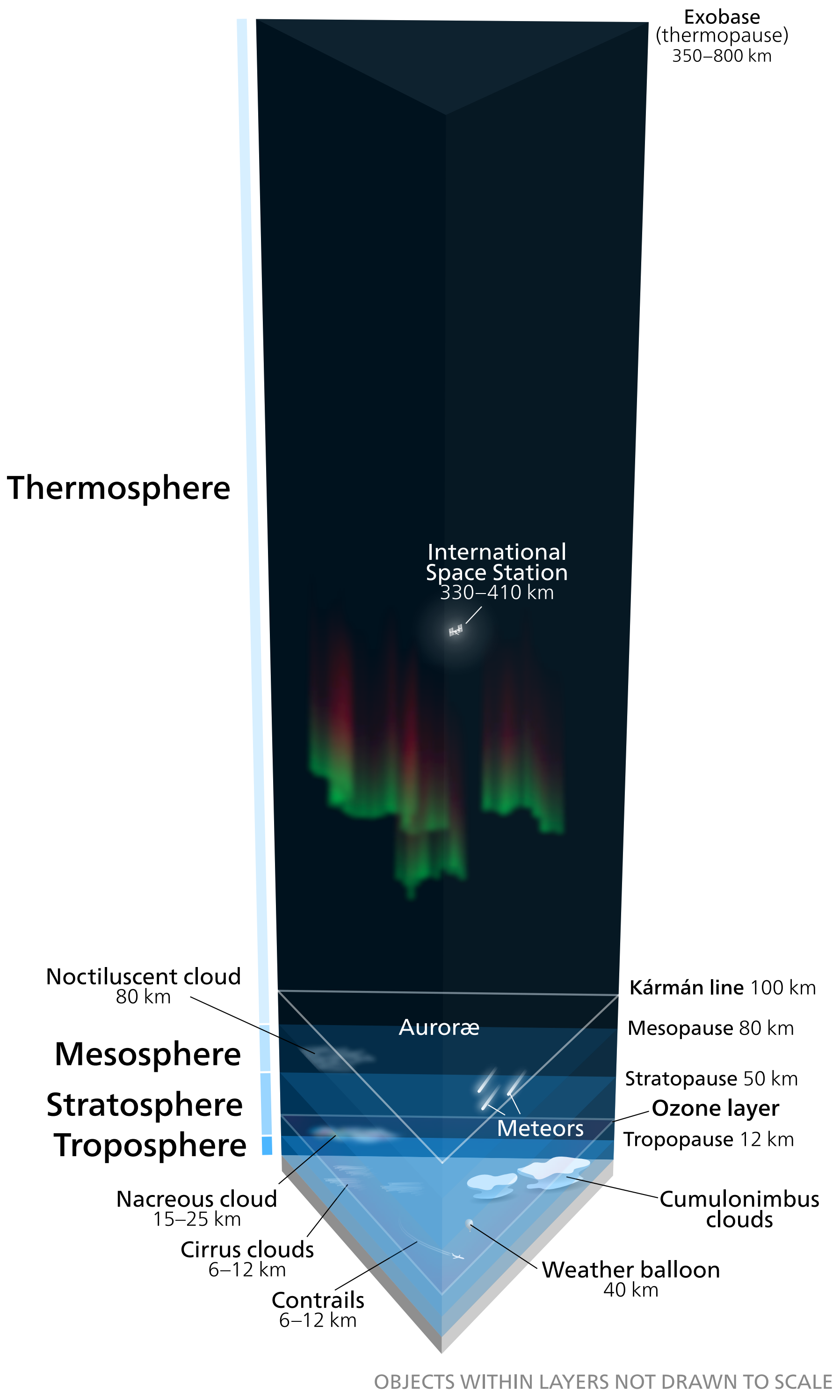


Figure 2.1: A cross section of Earth's atmosphere, showing atmospheric layers, their boundaries, and approximate locations of some types of clouds. Kelvinsong, available under CC-BY-SA 3.0 [13].

2.1.1 Cloud formation

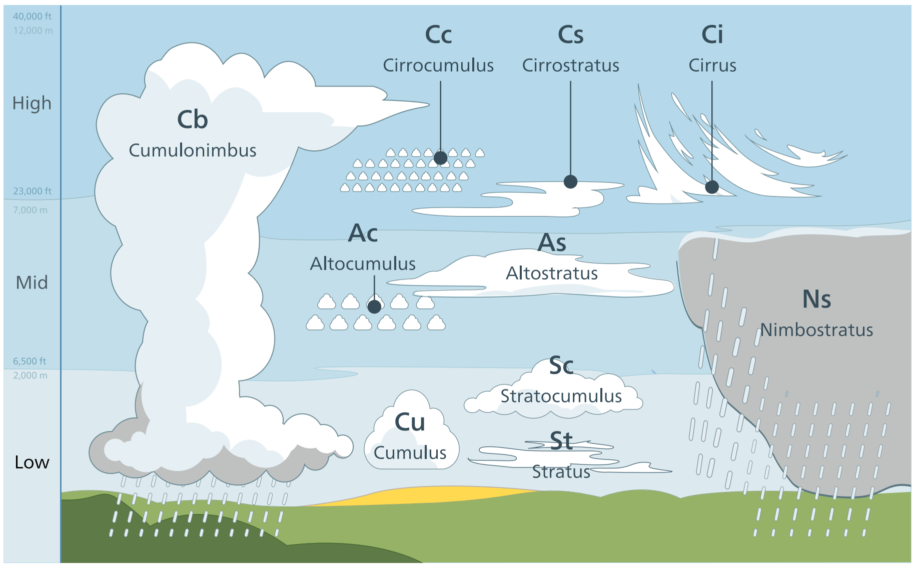


Figure 2.2: Common cloud types in planet Earth's atmosphere. Valentin de Bruyn / Coton, available under CC-BY-SA 3.0 [23].

When air is heated by the sun it rises upwards into the thinner parts of the atmosphere, potentially carrying water molecules along. In the lowest layer of the atmosphere – the troposphere – temperature decreases as height increases, and the air is saturated with water, at which point water droplets will start to condense and a visible cloud forms. Clouds, however, are not purely chaotically indistinct forms. See Figure 2.1 on the preceding page for an overview of the layers of the atmosphere.

Cloud types were classified and given Latin names by Luke Howard in 1803 [11]. Seven cloud types were listed in Howard's *Essay on the Modifications of Clouds* and described as follows:

- Cirrus: Parallel, flexuous, or diverging fibres, extensible by increase in any or in all directions.
- Cumulus: Convex or conical heaps, increasing upward from a horizontal base.
- Stratus: A widely extended, continuous, horizontal sheet, increasing from below upward.
- Cirrocumulus: Small, well-defined roundish masses, in close horizontal arrangement or contact.
- Cirrostratus: Horizontal or slightly inclined masses attenuated towards a part or the whole of their circumference, bent downward, or undulated; separate, or in groups consisting of small clouds having these characters.
- Cumulustratus: The Cirro-stratus blended with the Cumulus, and either appearing intermixed with the heaps of the latter or superadding a wide-spread structure to its base.

- Nimbus (Cumulo-cirro-stratus): The rain cloud. A cloud, or system of clouds from which rain is falling. It is a horizontal sheet, above which the Cirrus spreads, while the Cumulus enters it laterally and from beneath.

In addition to these types, the cumulonimbus, stratocumulus, altocumulus, and altostratus types were added; see Figure 2.2 on the preceding page. The extremely low-density noctilucent ("night shining" in Latin) clouds were only discovered late. Contrails are an artificial cloud type formed by the condensation following the jet stream of modern passenger aerocraft.

2.1.2 Light interactions with the atmosphere

As light enters the atmosphere, some of it is transmitted with its incident direction preserved while another portion is absorbed by the air and the last portion is scattered in all directions. This section introduces physical formulae for calculating the respective ratios of these quantities of light; these are key to reproducing visuals reminiscent of the actual atmosphere.

While transmittance and absorption are assumed to be non-directional (isotropic) for the purposes of this thesis, scattering is strongly dependent on the angle between the view vector – the direction from the lit location and to the camera – and the light vector – the direction from the lit location and to the light source. A pair of *phase functions* describe these direction-dependent (anisotropic) relations as distributions of the probability of light being scattered by a given angle away from its incident direction; see Figure 2.3 on page 16 for an illustration of these distributions over angles.

Given an atmosphere-relative location $\mathbf{p}$, the amount of light reaching $\mathbf{p}$ that will be scattered in a direction θ radians away is proportional to both a *phase function* of θ and a *scattering coefficient*, which is a function of the density of the relevant type of atmospheric constituent at $\mathbf{p}$ and the light wavelength λ. These functions differ depending on the scattering type. The two scattering types and their respective functions are given below (with the scattering coefficients of a cloudless sky, i.e. where the density at $\mathbf{p}$ is a function of the height above sea level $h = ||\mathbf{p}|| - R$, where R is the planet radius).

Rayleigh scattering

Most of the atmosphere consists of small molecules of oxygen and nitrogen, whose individual sizes are far below the scale of the wavelengths of visible light (i.e. approximately 380 nm to 740 nm). Light scattering due to such particles is called *Rayleigh scattering*.

The Rayleigh scattering coefficient β^s_{R} and phase function P_{R} as used by Bruneton and Neyret [3]

$$P_{\mathrm{R}}(\mu) = \frac{3}{16\pi}(1 + \mu^2) \tag{2.2}$$

$$\beta^S_{\mathrm{R}}(h, \lambda) = \frac{8\pi^3(n^2 - 1)^2}{3N\lambda^4} e^{-\frac{h}{H_{\mathrm{R}}}} \tag{2.3}$$

where $\mu = \cos\theta$, and θ is the angle between the incident direction and the outscattered direction, h is the height above sea level, n is the index of refraction of air, N is the molecular density at sea level, λ is the wavelength of the light, and H_{R} is the scale height of the Rayleigh density distribution.

As can be seen in equation 2.3 on the facing page, Rayleigh scattering is inversely proportional to the fourth power of the wavelength, which means different colours aren't scattered to equal extents. In particular, short wavelengths – such as blue – are scattered much more than long wavelengths – such as red – which explains both the daytime bluish tones of the atmosphere as well as the redder appearance of sunrises and sunsets; when light travels through longer and denser portions of the atmosphere, significantly less blue and more red light remain in the sun direction.

Mie scattering

Light scattering due to particles of a size similar to the wavelengths of light is called *Mie scattering*. The sizes of water droplets result in their aggregates (namely clouds, mist, and fog) being visually characterised by Mie scattering.

Mie scattering theory formulates a highly complex phase function that is impractical to evaluate directly. The Mie scattering coefficient β_{M}^{S} and phase function approximation P_{M} (the Cornette-Shanks phase function) as used by Bruneton and Neyret [3]:

$$P_{\mathrm{M}}(\mu) = \frac{3}{8\pi}\frac{(1-g^2)(1+\mu^2)}{(2+g^2)(1+g^2-2g\mu)^{3/2}} \tag{2.4}$$

$$\beta_{\mathrm{M}}^{S}(h,\lambda) = \beta_{\mathrm{M}}^{S}(0,\lambda)e^{-\frac{h}{H_M}} \tag{2.5}$$

where H_{M} is the scale height of the Mie density distribution, the parameter $g = 0.8$, and the other variables are the same as for the Rayleigh functions. Observe that Mie scattering's wavelength-dependency is independent of height, which means it can be evaluated once for all heights and then used as a constant factor. Additionally, these constant factors are in fact wavelength-independent; this gives the typically mostly white or grey appearance of clouds in daytime.

The Mie phase function has a very strong forward lobe: it scatters most of the light in the forward direction, and only small fractions in all other directions. Visually this emphasises the edges of clouds that (nearly) occlude the sun with distinct rims of heightened brightness, giving rise to the renowned *silver lining* visual characteristic; see Figure 2.4 on page 17 for an example.

Ozone absorption

In addition to Rayleigh scattering by several smaller molecules and Mie scattering due to larger droplets, ozone plays a noticeable role in atmospheric lighting by absorbing some light (particularly a portion of the green light).

The ozone density distribution is modelled as linearly increasing from a height of 10 km to 25 km and linearly decreasing from 25 km to 40 km. The ozone absorption is proportional to the ozone density and a wavelength-dependent coefficient $\beta_{\mathrm{O}}^{\mathrm{Ex}}(\lambda)$ but independent of the direction (i.e., it is isotropic).

The Beer-Lambert law

Some light is scattered away from the incident direction and more is absorbed by water droplets and ozone molecules, but there is a remaining portion that will not be deterred.

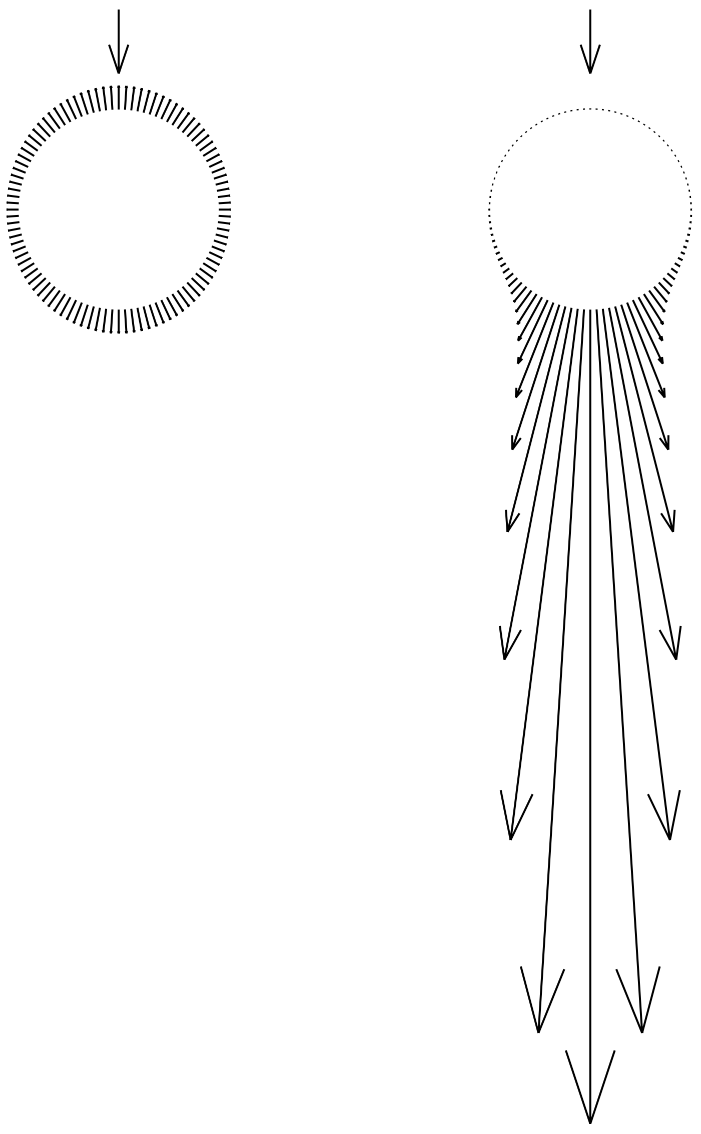


Figure 2.3: Visualisation of Rayleigh (left) and Mie (right) scattering phase functions: light coming from above is scattered in all directions; the arrow lengths are proportional to the probability of scattering in the arrow angle. No light scattering at all would be shown as a single straight arrow to the bottom, as incident light would travel on without changing direction. While Rayleigh scattering is almost evenly distributed in all directions, Mie scattering doesn't scatter as much away from the incident direction (though it is still noticeably different from no scattering at all).

Observe that these are cross sections: at first glance it may seem like the Rayleigh distribution is "too small", but this is only because the full distributions are actually over all three-dimensional directions (which could be visualised as the surface of a three-dimensional sphere).

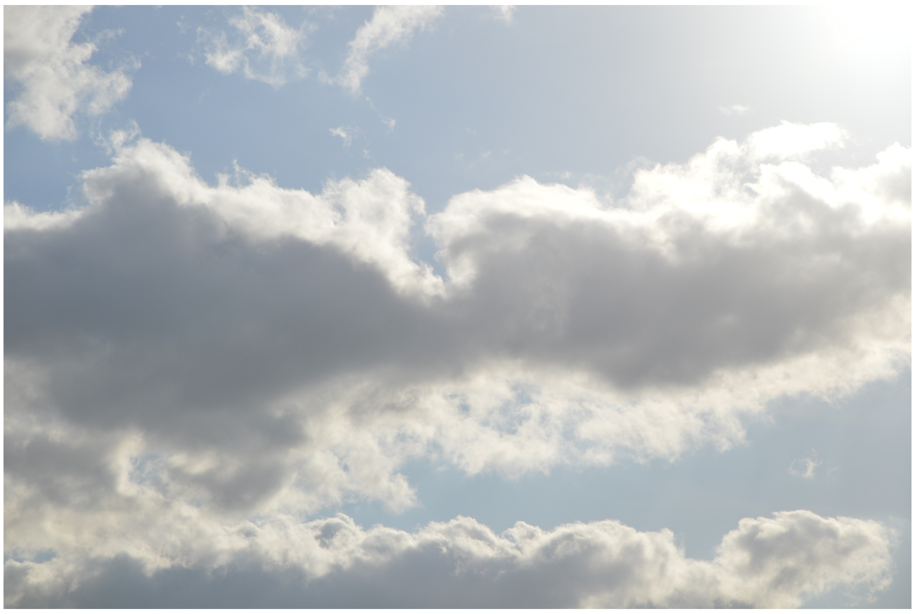


Figure 2.4: A photographed cumulus cloud formation close to the sun in the image, showing the bright silver lining effect on its edges due to the incident direction (towards the sun) and the view direction (from cloud to camera) being closely aligned and thus within the forward Mie scattering lobe.

The ratio of light continuing on the incident direction to the light intensity first entering the atmosphere (or, more generally, any participating medium) is called *transmittance*. Transmittance can be formulated as a function of optical depth in the form of the Beer-Lambert law (also called Beer's law) [26]:

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

2.2 Computer graphics

Real computer hardware cannot store real numbers with infinite precision nor directly manipulate the states of so many particles as to match reality one-to-one (not to mention the limitations of physical models and possibly unknown aspects of nature yet to be discovered and understood); to recreate the visual properties of the atmosphere, both physical light and the actual atmosphere have to be represented numerically in some capacity. To this end, a colour model is introduced, and a spatial partitioning structure is motivated and described.

2.2.1 Colour

Actual light can be composed of any combination of wavelengths on a spectrum; representing this accurately is considered prohibitive for most real-time rendering, due to the memory and computational costs associated with handling a more comprehensive spectrum for every pixel. Consequently, colours are typically reduced to only three wavelengths – red, green, and blue – which correspond to the most common types of photoreceptor cells (cones) in the human eye. In 1931, the International Commission on Illumination (CIE) defined the CIE 1931 colour spaces [16], which form a numeric basis for the wavelength-dependent responses of these cones cells. Other colour spaces – such as sRGB and ACEScg – can be transformed into the CIE 1931 colour spaces (see Figure 2.5). Many common monitors only support RGB colour triplets representing values in the sRGB colour space.

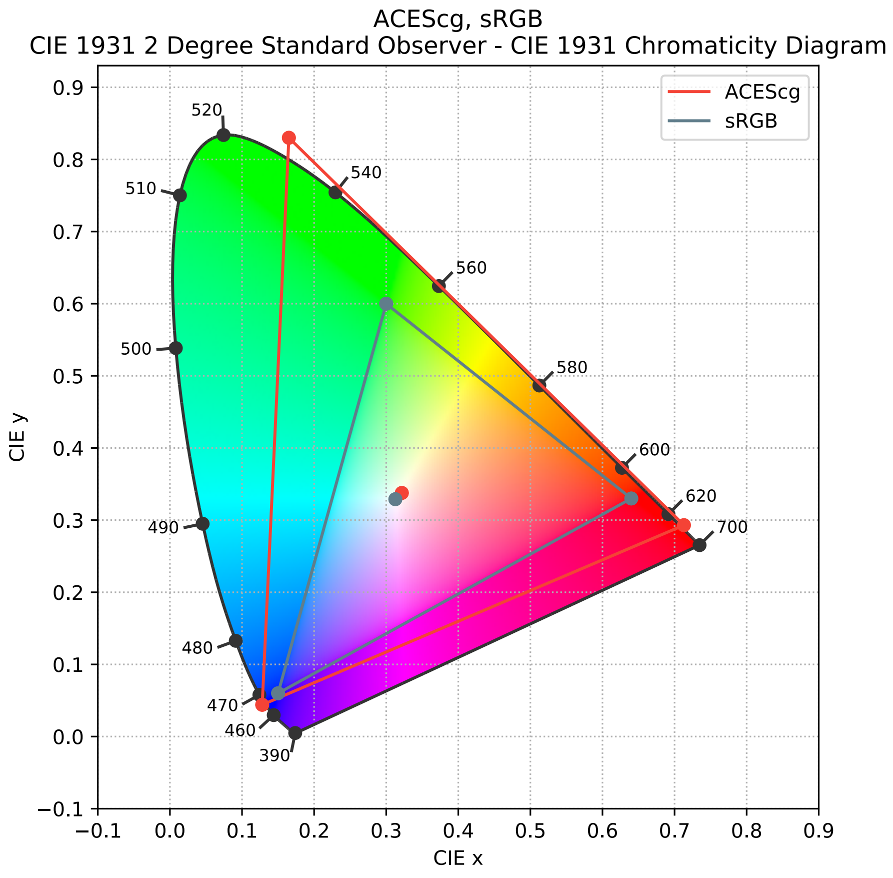


Figure 2.5: The ACEScg and sRGB colour spaces shown in the CIE 1931 chromaticity space (image created with the Python `colour-science` package). The axes are the normalised X and Y coordinates, spanning all chromaticities; the unshown third axis corresponds to brightness, which is orthogonal to chromaticity.

2.2.2 Dynamic range and tone mapping

In reality, there is in principle no limit to the power of a light source and thus no absolute upper bound on the amount of energy in the emitted light. However, computer monitors can only display a subset of this vast dynamic range (and the human eye can only adapt so far to darkness or brightness). Thus software renderers often compute images initially in a high dynamic range (HDR) corresponding more closely to physical values and the possibly huge or minuscule magnitudes allowed by physics, and then convert the images to a low dynamic range (LDR) for display only after all physically-based effects have been accounted for. This can be seen as a virtual equivalent of the physical exposure taking place in physical cameras.

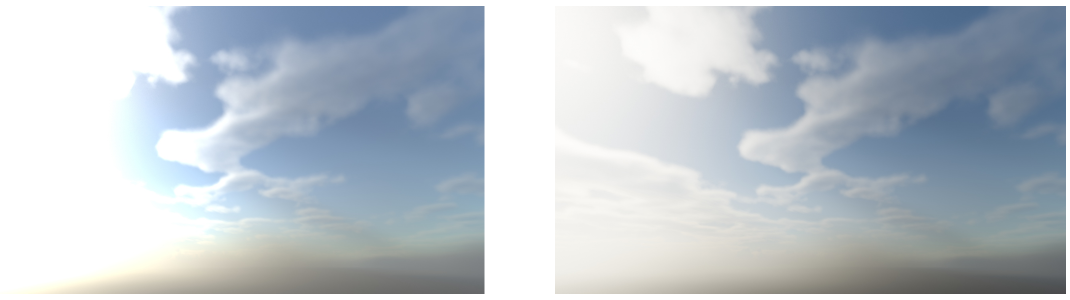


Figure 2.6: The difference between just clamping colour intensities to the low dynamic range (left) and applying the ACES tone mapping operator (right).

The process of converting from HDR to LDR is called *tone mapping*, and functions mapping from HDR values to LDR values are called tone mapping operators. There are many tone mapping operators, among them one that is part of the Academy Color Encoding System (ACES), a colour management standard widely used in computer graphics. The ACES tone mapping operator is based on a spline fitted by experts reviewing the behaviour of the operator; it is intended to emulate real film, and often avoids too abrupt oversaturation in the brighter areas of images while making for dramatic colour hues (see Figure 2.6).

2.2.3 Voxels

Pixels (picture elements) are values sampled in a uniform grid on a 2D plane; analogously, *voxels* (volume elements) are values sampled in a uniform grid on a 3D plane. Depending on usage, voxels can be sampled in the centres of grid cells – and perhaps rendered as cubes occupying the grid cells – or rather sampled at the corners of intersecting grid lines and interpolated in-between to give a continuous value distribution over the entire volume; see Figure 2.7 for an illustration of the different sampling patterns, and Figure 2.8 on the next page for the difference due to linear interpolation as opposed to no interpolation.

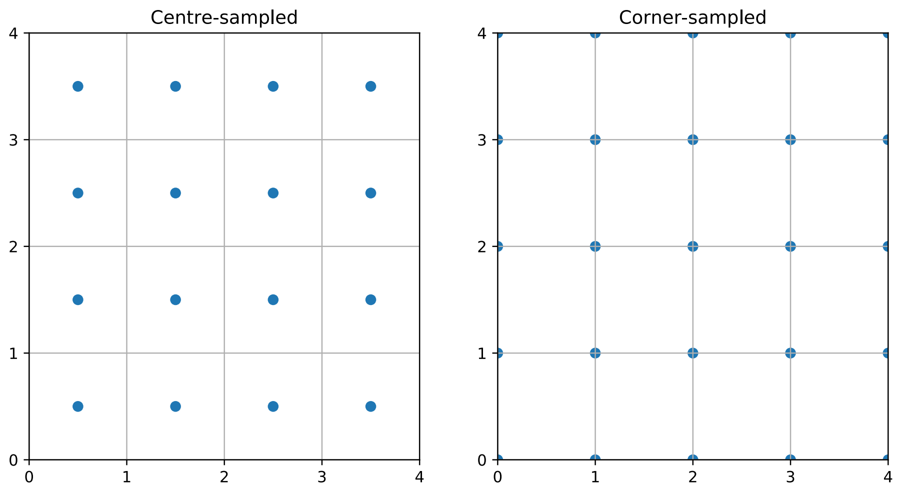


Figure 2.7: A 2D cross section of a 3D voxel volume. Voxels can be either sampled at grid cell centres (left) or at grid cell corners (right).

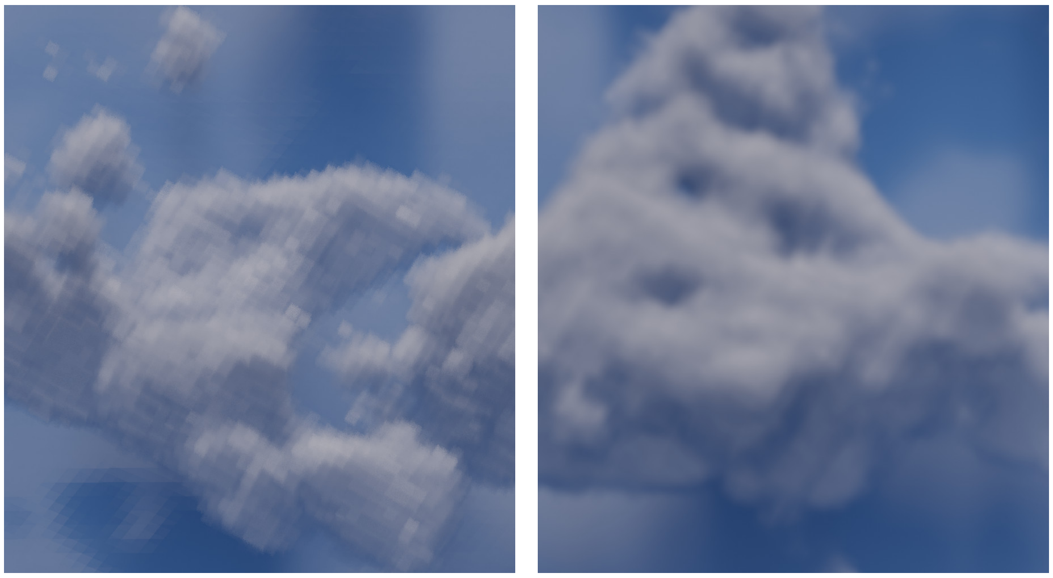


Figure 2.8: A voxel volume rendered with no interpolation between voxel values (left) and linear interpolation between voxel values (right).

2.2.4 Octree

In theory, a single giant voxel matrix of densities can represent any density distribution in space. Yet, in practice this requires far too much memory for all but the simplest and smallest of scenes.

A quick foray into mental arithmetic to show the infeasibility of representing an atmosphere density distribution with a straightforward 3D matrix: the Earth has a radius of about 6371 km, and its atmosphere extends roughly 100 km farther out. For simplicity, let's say the atmosphere diameter is roughly 10^4 km (and note that rounding down errs on the side of caution in the following reasoning; it means the actual requirement would be even higher). From this follows that more than $(10^4)^3 = 10^{12}$ values are required to have one value per cubic kilometre in a 3D matrix encompassing the entire atmosphere. Since this is still quite coarse, we may multiply the estimate by perhaps another 10^3 (if e.g. features down to 100 m should be representable). With at least one byte per voxel, this is on the order of a million gigabytes of data, far outstripping today's GPU memory capacities of just a few tens of gigabytes on the highest end (and even storage space of several terabytes).

Of course, a large fraction of this volume is the interior of the planet, and much of the rest is so far from the planet as to be in outer space, wherein the atmosphere doesn't exist at all; a solution with density values bounded more closely to the surface of the planet would be less wasteful. The Earth's surface area is roughly $500 \times 10^6\,\text{km}^2$; if we want to represent up to 10 km tall clouds, and once again multiply by 1000 to account for cloud detail sizes down to 100 m, this would require somewhere above 10^{12} bytes, i.e. 1000 GiB (not accounting for the higher "layers" having an even greater area than the surface area at ground level, as the outer shells have larger radii). Even this is well beyond current hardware.

But for many scenes, even a uniform grid conforming closely to the surface is highly wasteful: there would be pockets of emptiness, perhaps others of constant density, and only

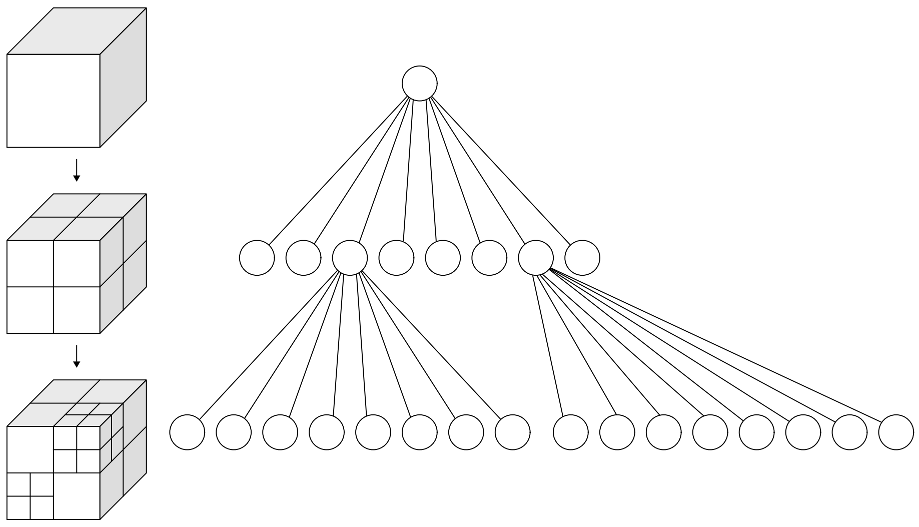


Figure 2.9: A cubical volume split recursively in octants, as by an octree, which is also represented with the per-octant nodes of various sizes and their child nodes. WhiteTimberwolf, available under CC-BY-SA 3.0 [25].

some regions occupied by actually varied values of interest. Additionally, and more significantly, regions that are more distant from the current camera location can make do with relatively coarser details than those regions that are nearer the camera. It would thus be best if the emptier and more distant regions could occupy less memory, and this can be achieved with the use of a space partitioning structure, such as an *octree*.

An octree is a tree data structure representing a division of a 3D space in 8 equally-sized octants, each represented with a *node* in the octree. Each octant may be split into 8 smaller octants (corresponding to 8 smaller nodes added to the tree), and so on recursively [27] (see Figure 2.9). If only those nodes which contain information of interest are split to high depths and high detail, resources may be conserved for where they are needed most, since the less interesting regions are only represented at lower resolutions. When used in the manner described, such an octree is known as a sparse voxel octree [28].

2.2.5 Ray marching

Volumetric data can be straightforwardly rendered by starting from a chosen ray origin location and then stepping along the ray direction and sampling values at these discrete step locations (see Figure 2.10 on the next page). The number of steps (which is inversely proportional to the step size) controls a tradeoff between quality and performance: the fewer the steps, the faster the computation; but fewer steps may result in missing the thinner of details, and can also give rise to visually unpleasant banding (apparent quantisation of colour and lighting).

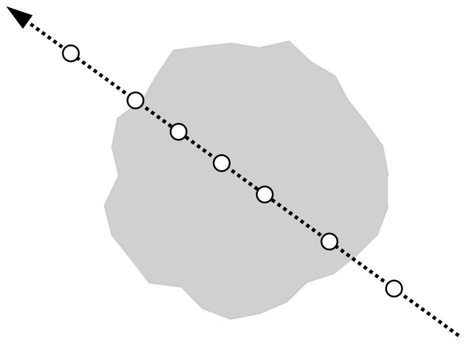


Figure 2.10: Ray marching: steps – whose sample locations are shown as circles – are taken along a ray, in this case intersecting a simple cloud shape. The step size does not have to be constant; the steps are unevenly distributed in this instance, with shorter steps being taken within the cloud, which is perhaps indicative of the interior of the cloud needing more consideration than the air outside it.

2.2.6 Graphics memory

Memory resources used for rendering on GPUs (Graphics Processing Units, also known as graphics cards) are allocated in one of two forms: as textures and as buffers. The former have special properties, while the latter are essentially ordinary arrays of arbitrary memory structures.

In computer graphics parlance, the word *texture* often has a specific meaning: a low-dimensional (typically 1 to 3 dimensions) matrix of elements. The individual texture elements – *texels* – can and often do represent colours; however, in general the texel data may represent arbitrary values, the exact meaning being up to the intended use and the programmer interpretations encoded in programs. A texture spans a unit coordinate space, and values sampled on this space will be interpolated from the texels on the grid corners around the sample location (if the appropriate texture filtering mode is set), which can be used to greatly accelerate many graphical algorithms while avoiding the visually blocky look that comes from not interpolating and rather snapping to the nearest texel.

2.2.7 Procedural noise

A short chain of mathematical functions applied to input coordinates can result in many shapes, ranging from simple and abrupt step functions to smooth gradients and periodic sinusoids, among a vast space of potential combinations. However, most of these are clearly unnatural-looking in many contexts where one would want a result that only on average resembles such a pure function, while smaller perturbations and fluctuations are desired. In these cases, which include various forms of procedurally generated clouds in computer graphics, a well-tuned amount of seeming randomness can make a critical difference.

Perlin noise is a pseudo-random continuous scalar function in one or more dimensions, invented by Ken Perlin in 1985, based on interpolating values from the closest grid points in an N-dimensional cube analogue [15]. Given input coordinates, it returns values that appear

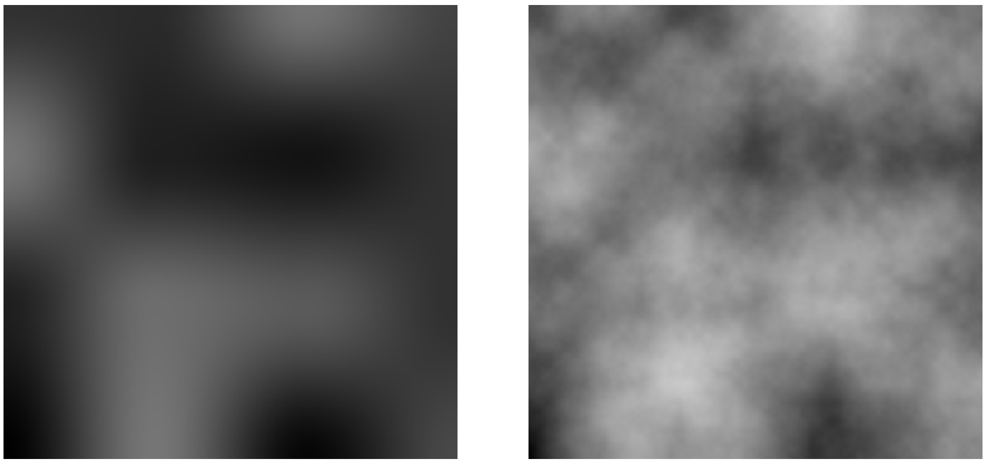

(a) 1 octave of Perlin noise. **(b)** 6 octaves of Perlin noise.

Figure 2.11: A single octave of two-dimensional Perlin noise (left) besides a sum of 6 octaves of two-dimensional Perlin noise with persistence = 0.5 and lacunarity = 2 (right). These images were created with code samples from The Book of Shaders [24].

to be random over larger scales yet are continuous.

The original Perlin noise algorithm suffered from directional artefacts (aligned with the cardinal axes) and poor computational scaling to higher dimensions (proportional to N^2, where N is the number of dimensions). In 2001, Ken Perlin created simplex noise, a similar but improved noise algorithm based on simplexes – the N-dimensional generalisation of triangles and tetrahedrons – rather than the hypercube grid in Perlin noise [8]. The number of values interpolated in a simplex noise evaluation scales linearly with N.

By itself, a single evaluation of a noise function gives an uninterestingly simple (smooth) continuum. By summing multiple evaluations – *octaves* – at exponentially higher multiples of the input coordinates and with exponentially decreasing coefficients, a more natural-looking texture can be achieved. See Figure 2.11 for examples.

Such a sum of noise evaluations approximates *fractal Brownian motion* (fBm):

$$\mathrm{fBm}(\mathbf{p}) = \sum_{n=1}^{o} a^n \,\mathrm{noise}(l^n \mathbf{p}) \tag{2.7}$$

where $\mathbf{p}$ is a coordinate vector, o is the number of octaves, a is the persistence, and l is the lacunarity. The number of octaves controls how high the noise resolution gets and thus how fine details can be produced, while persistence characterises the amplitude falloff with higher frequencies, and lacunarity determines how quickly the frequency increases.

There are other common types of procedural noise functions (among them cellular noise, which features in multiple cloud renderers' 3D textures (e.g. [2] and [14])), but this thesis relies mostly on a combination of fBms based on Perlin noise. One fBm value can for example be used to modulate or mask (threshold) another fBm evaluation at a different coordinate offset, to generate areas whose boundaries are of lower frequencies but where the interior details are of higher frequencies.

Chapter 3
Related work

Cloud rendering – real-time and otherwise – is a well-researched area. In this section the prior works which this thesis is directly inspired by and based on will be revisited, and the aspects most relevant to this work will be noted and commented on.

3.1 GigaVoxels (Crassin, 2011)

In 2011, Cyril Crassin presented GigaVoxels [4], a system for representing objects using voxels in multiple levels of detail in real-time. In GigaVoxels, objects are represented with octrees where the leaf nodes correspond to voxel *bricks*, cubical matrices of low resolution. By not splitting the octree nodes all the way down to individual voxels, and rather choosing a suitable brick size, memory coherence (and a slightly fewer number of octree levels to traverse) contribute to better render performance and adaptive level of detail management as bricks can be loaded or unloaded efficiently.

Principles from and based on the GigaVoxels system are central to the atmosphere representation and level of detail management implemented in this thesis.

3.2 Precomputed atmospheric scattering (Bruneton and Neyret, 2008)

In 2008, Eric Bruneton and Fabrice Neyret introduced methods of precomputing atmospheric transmittance and light scattering into lookup tables and then interpolating the table values to render an atmosphere efficiently, accounting for Rayleigh and Mie scattering due to density functions dependent only on height [3]. In 2017, Bruneton reimplemented the code to be more readable and consistent in variable and function names. Absorption due to the ozone layer was also added in this version, further improving the visual believability of the results [6].

Parts of Bruneton's and Neyret's method – specifically the precomputation of clear-sky transmittance and single scattering – are used for significantly optimising the algorithm in this thesis.

3.3 Ray marched texture combinations

Predominantly ground-based views of cloudscapes can be rendered by ray marching some weighted combination of tiled and scaled noise textures. Multiple implementations and techniques build on this principle.

In 2016, Rikard Olajos included Gaussian towers (multivariate normal distributions centered on pickable coordinates) to control the locations and dimensions of the biggest cloud features when computing a low resolution cloud density 3D texture for use in combination with a tiled 3D texture [14]. This method enables some control of cloud feature placement, but restricted to the confines of the tiled 3D texture. This type of constraint – and the need for tiling cloud textures – was avoided in my work.

The Nubis cloud system in the Decima video game engine developed by Guerrilla Games [2] (displayed prominently in the 2017 video game Horizon Zero Dawn) is another example of tiling a low resolution 3D texture modulated by a 2D texture to control the cloud cover. Hand-adjusted gradients are used to achieve specific cloud types, such as cumulus or cumulonimbus. Notably, Nubis makes use of a simple yet visually effective heuristic for quickly estimating the indirect light scattering in the clouds, which was also used in this work.

In 2018, Fredrik Häggström conducted a thorough investigation of many aspects of the now typical ground-view cloud rendering based on combining a few coverage textures, height gradients dependent on desired cloud type, and a tiled low resolution detail volume texture. This solution did not implement higher cloud layers, but noted that they could be added at an additional performance cost [9]. In contrast, the algorithm presented in this thesis avoids increasing render algorithm complexity (to account for more cloud layers and types) by separating cloud generation and cloud rendering.

The atmosphere implementation in the video game Red Dead Redemption 2 by Rockstar Games also renders clouds by ray marching a combination of low resolution noise textures and masks. In addition, it integrates light contributions from fog and mist (close to the camera) in a low resolution frustum-aligned 3D texture [7].

The conventional ray marched approaches work well for camera locations relatively close to the ground, but cannot be easily extended to elevations significantly above the clouds or in outer space. Avoiding this limitation was a major motivation for this thesis, and a significant feature of the atmosphere renderer Mulen developed as part of it.

3.4 Voxel-space lighting

In 2019, Carlos Jiménez de Parga Bernal Quirós published a PhD thesis including an extensive survey of previous cloud rendering techniques, an algorithm for real-time physically-based fluid dynamics modelling of clouds, and a method of computing cloud lighting in cloud-space voxel grids [12]. This method was used for scales around individual clouds, but not extended to representing planetary-scale cloud formations.

This particular instance of optimising voxel-space lighting by not fully recomputing it per-frame and per-pixel helped directly inspire the approach to cloud lighting in this thesis.

Chapter 4
Algorithm

This chapter provides an outline of how the atmosphere renderer Mulen functions, beginning with the core concept, continuing onto the parts which are precomputed once, and moving onto the continuous operations repeated until the program is shut down.

4.1 Core concept

Mulen represents an atmosphere as a sparse voxel octree. The octree nodes are created in groups of eight, beginning with a root node group enclosing the entire octree volume; when a node is split, a new node group become its child nodes. Each node contains not individual voxels but rather bricks of 8^3 voxels, as in GigaVoxels [4] but in this case focused only on representing cloud formations. Voxels are corner-sampled rather than centre-sampled to facilitate linear interpolation between voxels without having to access neighbouring nodes and bricks (which helps avoid obvious visual blockiness).

Voxel data Each voxel has both a Mie density value and a shadow value. The density is computed first, according to an algorithm henceforth referred to as a *generator*. Lighting is then partially precomputed by ray marching once per voxel through the atmosphere octree, accumulating shadow contributions from interpolated density values between the origin voxel and either the end of the atmosphere or the intersection point with the planet sphere (whichever comes first), and storing the resulting per-voxel shadow value.

Parameters Two parameters control the quality level: *the memory budget*, and *the maximum allowed octree depth*. At program initialisation, voxel bricks and octree node storage and other auxiliary structures are dimensioned to occupy memory up to the desired memory budget; thus, the amount of memory used corresponds directly to the value of this parameter. The maximum depth determines to how finely-grained levels the nodes can be split and thus how small individual voxels may become. Together, the parameters bound the amount of computational time used for rendering; by adjusting them, quality and performance can be traded freely for one another.

Stages Mulen's computations are divided in two stages: initial precomputations – of both transmittance and first order light scattering due to the base height-dependent Rayleigh and Mie density distributions – followed by continuous operation; the latter is composed of updates to the atmosphere structure and spatially varying Mie density distribution stored as voxel data, as well as the rendering of these from a camera's view into the atmosphere.

4.1.1 Octree descent

A given location $\mathbf{p}$ within the octree volume lies within the space enclosed by one of the eight root nodes and potentially several of its child nodes and their child nodes in turn, recursively, the number of nodes equal to the depth of the octree at $\mathbf{p}$. To find the smallest (i.e, deepest-level and most finely detailed) node enclosing $\mathbf{p}$, the octant within which $\mathbf{p}$ lies is determined by subtracting the node centre location from $\mathbf{p}$ and examining the signs of each resulting coordinate value, and the child node corresponding to the octant becomes the current node; this process is repeated until a leaf node is reached and there are no more child nodes to iterate down to.

4.1.2 Octree ray marching

Octree ray marching is conducted with a double-nested loop: in the outer loop, an octree descent at the current step sample location (which is initially the ray origin) gives an octree node; in the inner loop, the ray is marched until the next step sample location is outside the node, at which point the inner loop breaks to a new iteration of the outer loop. If the current step sample location is outside the octree, the ray marching is terminated.

4.2 Precomputation

Precomputation of transmittance and first order light scattering are accomplished as in Bruneton's precomputed atmospheric scattering re-implementation [6].

4.2.1 Transmittance

The transmittance of a ray through the atmosphere considering only the Mie and Rayleigh density distributions as functions of height – i.e, without considering the voxel Mie density – is precomputed as in Bruneton's 2017 implementation [6], parameterised by height above sea level and the cosine of the angle between the ray direction and the ray origin to planet centre vector.

To also account for the regionally-variable decrease in transmittance due to the cloud Mie density when computing shadowing per-voxel and when rendering per-pixel, the precomputed transmittance is multiplied with the per-frame numerically iterated regional transmittance (as cloud cover can only decrease – and not increase – the ratio of light blocked by a distance through the atmosphere).

4.2.2 First order light scattering

First order light scattering in a clear part of the atmosphere (i.e., where the voxel Mie density is not considered) is also precomputed as in Bruneton's 2017 implementation [6], integrating Mie scattering, Rayleigh scattering, and ozone absorption (see Equation 2.2 on page 14, Equation 2.3 on page 14, Equation 2.4 on page 15, and Equation 2.5 on page 15). The precomputation is parameterised by height above sea level, the cosine of the angle between the ray direction and the ray origin to planet centre vector, the cosine of the angle between the sun direction and ray origin to planet centre vector, and the cosine of the angle between the ray direction and the sun direction.

4.3 Continuous operation

Update and render loops iterate continuously until the program is exited, executing distinct sequences of passes (see Figure 4.1 on the next page). While one render iteration is executed once per frame, the update loop executes at a lower frequency to conserve computational resources. This means the render iterations do not have fully up-to-date voxel data, but since the clouds are naturally slow-moving the renderer interpolates linearly between the last two completed update iterations' voxel data to approximate ideal per-frame Mie density distributions. Since the renderer thus needs access to two full sets of voxel data while the updater is constructing a new set in parallel, three sets of voxel data need to be kept in memory simultaneously.

4.3.1 Update

Before anything more than a clear sky can be rendered, the Mie density distribution and the resulting shadows have to be computed. Mulen accomplishes this in a series of passes interleaved with rendering over multiple frames so that the slow generation work can be redone less frequently than the per-frame rendering. All voxel data is re-generated once per second while rendering interpolates between the last two completed sets of voxel data, meaning a total of 3 sets of octree and voxel data are in memory simultaneously: two previously finished complete sets current for times up to two and three seconds before the present time, and a new state being computed over the current update iteration until it's finished, at which point the memory used for the oldest state becomes the destination for the next upcoming update iteration results.

Initialisation At the start of an update iteration, a split priority number is computed for each octree node, proportional to the node's physical size and inversely proportional to its distance to the camera. The split priority numbers of all leaf nodes (along with the node indices) are inserted into a split priority list in descending order, and the split priority numbers of all leaf nodes' parents are inserted into a merge priority list in ascending order. The split priority list is iterated from highest to lowest priority; if the lowest-priority node in

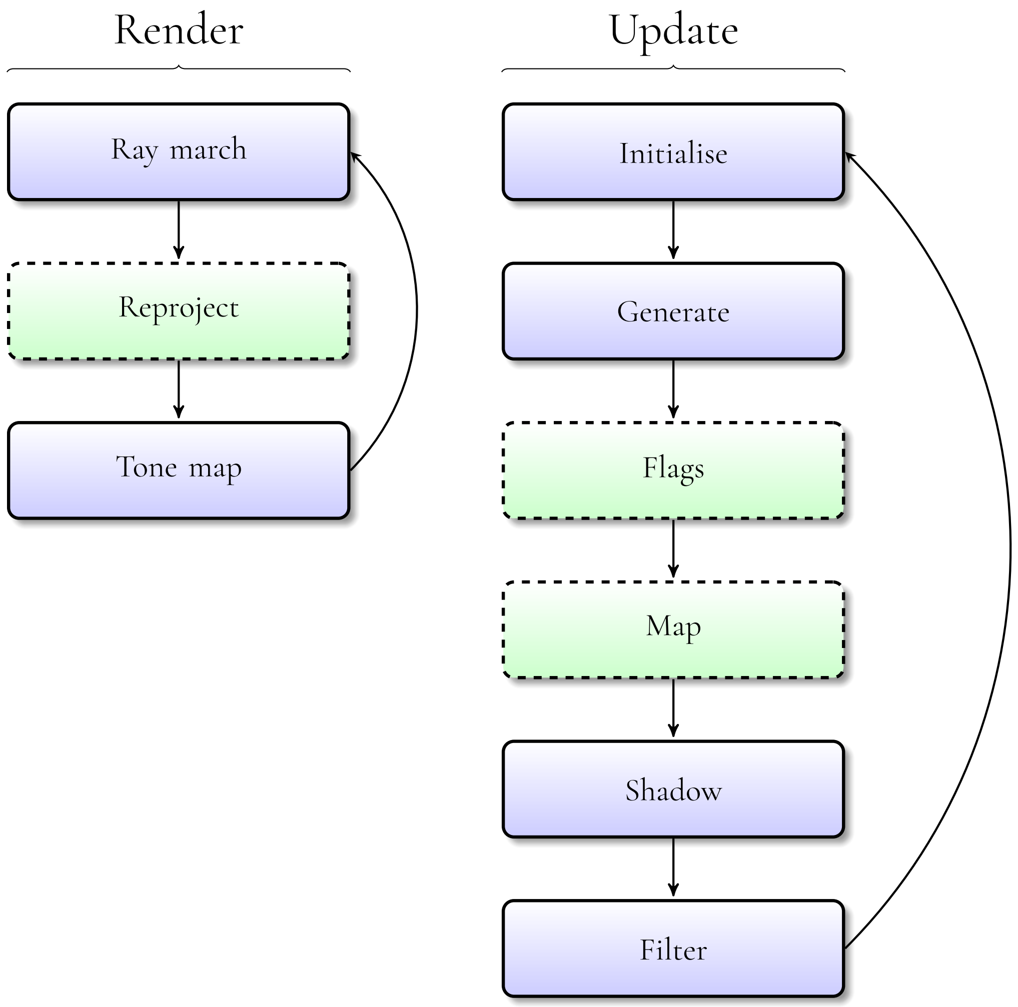


Figure 4.1: The constituent passes in single iterations of the continuous stages – update and render – with arrows showing the flow of time. While passes within a stage proceed sequentially, a complete render iteration may conceptually execute in parallel with just parts of one or more update stages. Optimisation passes (see Section 5.4 on page 37) are shown with a green background and dashed borders.

the merge list has a greater priority than the current split priority node, the loop terminates. Otherwise, the current merge priority node's child node group is merged – freeing a node group index and the associated memory slot and voxel brick – and the current node priority list node is split.

When a new update iteration is just finished and the next one is begun, rendering will start interpolating between the two sets of voxel data produced by the two most recently completed update iterations. To avoid render artefacts from interpolating between relative brick memory locations used for nodes which were newly split in the last completed iteration, the oldest brick data used for these node indices are initialised with interpolated brick data from the old parents of the newly split nodes. Since this update pass cannot be split over multiple render passes without risking artefacts caused by interpolating between different locations, it is computed over a single frame, and to decrease the time taken the maximum number of node splits allowed per-frame is capped to one tenth of the full node capacity allowed by the memory budget.

Density generation In the initial generation pass, per-voxel Mie densities are computed by per-voxel compute shader invocations. A procedural generation algorithm involving multiple octaves of Perlin noise and thresholding functions takes only the node position and size in octree space as input and creates a density distribution that exhibits three differently-masked fBm terms which were meant to loosely resemble stratus, cumulus, and cirrus layers.

Shadow pass In the shadow pass, a shadow ray is sent out from each voxel towards the light direction, accumulating density until zero is reached or the ray exits the atmosphere. This pass is responsible for the distinctive crepuscular rays (also known as god rays) and varying degrees of shadows in the clouds.

Shadow filtering Because the shadow pass outputs sharply aliased data, the shadow values are low-pass filtered by averaging over adjacent voxels before being stored in the second channel of the per-voxel brick data together with the already saved density values.

4.3.2 Render

Per-pixel computations through the octree give per-pixel light and transmittance values. The transmittance modulates potential light sources located behind the atmosphere (such as direct sunlight), and the sum of this light and the lighting scattered by the atmosphere is finally tone mapped.

Ray march Per-pixel octree ray marching integrate transmittance and inscattering as contributed to by both the per-voxel Mie densities and the base Rayleigh and Mie and ozone densities at step sample locations from the camera and through the atmosphere. The step size is proportional to the node size, so that smaller steps are taken where the voxel data is more detailed.

The precomputed per-voxel shadow values let this pass avoid branching ray marches, as the shadow data is already available and the transmittance factor due to a clear sky along shadow rays is retrieved from the precomputed transmittance table. The ray marching may stop early at per-pixel depth values from opaque objects rendered before the atmosphere, if there are any and if they occur before the ray leaves the atmosphere.

Tone mapping Finally, the per-pixel light values can be tone mapped so that they may be displayed on a typical low dynamic range display.

4.3.3 Higher-order light scattering

Mulen doesn't take into account general light scattering of orders higher than one (i.e. no light which would "bounce" more than once) from Rayleigh or Mie scattering in the air. However, higher-order light scattering inside the clouds is roughly approximated by adding an amount of light proportional to a function of per-sample density formulated by Andrew Schneider in the context of the Nubis cloud renderer in the Decima game engine [2]:

$$1 - e^{1-\text{density}} \tag{4.1}$$

Chapter 5
Implementation

This chapter delves into practical considerations and expounds upon technical choices in the implementation of the program. It lists the software libraries upon which Mulen relies, provides a brief introduction to the OpenGL graphics API as well as how it was used for the purposes of this thesis, and describes crucial optimisations for decreasing the computational cost of both updating (generating voxel densities and computing shadows) and rendering the voxel atmosphere.

5.1 OpenGL

OpenGL is a graphics API (Application Programming Interface) standard specifying a means to access functionality for issuing rendering and compute operations, typically accelerated with graphics hardware, i.e. GPUs. It is cross-platform, unlike the graphics API Direct3D, and not as low-level or recent as the cross-platform Vulkan API.

I chose to implement this thesis program in OpenGL partly because of my pre-existing experience with the API, and also because there seemed to be no need for locking Mulen to one platform. Vulkan was a conceivable alternative, but ultimately my personal familiarity with OpenGL took precedence. Mulen does not use any OpenGL extensions without equivalents in Vulkan, so a change of graphics API shouldn't be a very arduous task.

The graphics pipeline A *shader* is a program running on the GPU, typically in many – often millions – of conceptually parallel (i.e. simultaneous) invocations. Similarly to other graphics APIs, OpenGL defines a pipeline consisting of several stages which process some form of input by launching shader invocations for each input item. The traditional way of rendering begins its programmable part with the vertex shader (run once per vertex, e.g. three times for a triangle unless either of the triangle vertices is cached from an earlier invocation in the same draw call) whose output is rasterised onto pixels, each of which a fragment shader invocation is launched for. In-between these there are several optional shader stages (geometry shaders and tessellation shaders) and after the fragment shader there are several optional tests (the depth test, scissor test, and colour masking) [1].

For non-triangle-based passes, traditionally a fullscreen quad (two triangles covering all of the viewport) is launched. However, there is also the option of opting out of the vertex-based pipeline and instead using a compute shader, which is launched in a discrete number of workgroups, each the same per-shader 3D size of invocations [1].

Mulen mostly operates through compute shaders; vertex and fragment shaders are confined only to the placeholder rendering of a perfect sphere as planet backdrop and the final writing of colour values to the back buffer. Since a compute shader's workgroup size is three-dimensional, it is a good fit for voxel operations over three-dimensional bricks. For shaders operating on 2D images the third workgroup size component is 1.

5.2 Libraries

The following open-source software libraries were used in Mulen:

- GLFW [19] to create a window and handle input (from keyboard and mouse).
- glad [17] to load OpenGL functions.
- GLM (OpenGL Mathematics) [18] for vector, matrix, and quaternion structures and functions.
- Dear ImGui [22] for user interface (controls and information display of assorted continuous atmosphere and profiling data).
- LodePNG [20] to encode and save PNG image files (including embedded key-value string pairs to store and retrieve Mulen camera parameters in screenshot files).
- JSON for Modern C++ [21] to record and benchmark setups, and save benchmark measurements.

All libraries are cross-platform with support for operating systems such as Windows, Linux, and macOS.

5.3 Architecture

The atmosphere object is initialised with a set of parameters, including a GPU memory budget. The atmosphere object allocates textures and buffers sufficiently large to hold as many node groups and voxel bricks as possible within the memory budget constraint. On each render frame, the atmosphere is first updated and then rendered.

Modifications to the CPU octree are carried out by a separate updater thread, to ensure the once-per-second updates do not impact the framerate due to CPU load. In the initialisation stage, the updater communicates with the render thread to exchange a newly finished update iteration and transfer the parameters defining the next update iteration to be computed. The new CPU octree is copied to GPU memory in the update initialisation pass, before brick generation.

The brick data (i.e, per-voxel density and shadow value) are stored in a 3D texture with two 8-bit channels.

5.3.1 Updater

Atmosphere updates are split in three parts which are handled in different places: once-per-second updates to the octree are computed by a worker thread; the render thread receives once-per-second updates and consequently issues the OpenGL calls for uploading new octree data and recomputing densities and lighting over the course of one second; and compute shaders running on the GPU actually compute and store the new per-voxel densities and shadow values.

To avoid rendering an incomplete state (which could have shown as jagged artefacts where adjacent nodes in different density and/or lighting states would not match each other's border values), there are three sets of voxel data and octree in GPU memory simultaneously: one which is being generated during the current update iteration, and the most recent two prior states. Two old states are required so that smooth animation can be achieved by interpolating between these states over time, until a new state is completed and the oldest state is discarded so its memory can be reused for the next upcoming state.

5.3.2 Renderer

Mulen uses Stephen Hill's fit of the ACES tone mapping operator curve [10] for tone mapping light values before gamma correction and finally display.

5.4 Optimisation

Actual computer hardware is limited in both computational capacity and amount of memory; to leverage the GPU and attain sufficient speed, Mulen has to employ several optimisations (see Figure 4.1 on page 32). The most significant of these are elaborated in this section.

Brick flag computation After densities have been determined, the minimum and maximum densities per brick are computed and used to determine which bricks are empty (meaning all density values in the brick are equal to zero); a per-node bit flag in the GPU octree buffer is set to indicate empty bricks. These flags can later be used for optimising the shadow pass, as empty bricks can be skipped in one step instead of being traversed voxel by voxel.

Octree map To decrease the number of octree traversal iterations required per-node (in both the update shadow pass and the render ray march pass), an additional pass computes a low-resolution 3D integer texture containing the deepest octree nodes not bigger than the texture voxels throughout the entire octree volume. In this implementation, the texture size was set to 64^3, which means up to $\log_2 64 = 6$ iterations per traversal are saved by using the octree map.

Precomputed light scattering above cloud level Precomputed light scattering in Mulen is used above an arbitrarily defined cloud level (by default 25 km above sea level for the atmosphere parameters used in this thesis) to avoid long ray computations in volumes where there are no clouds. This introduces an error, as clouds below the cloud level would be able to shadow the upper layers of the atmosphere in regions close to the terminator (locations currently close to sunrise or sunset). This is ignored for simplicity.

Reprojection Per-pixel ray tracing of the atmosphere octree is highly computationally expensive. To increase speed manifold, only a lower resolution sparse portion of the full view is computed per frame, and these results are combined with a historic 2D texture of past results (and so on until the number of past frames as seen from the current one exceeds the downscale factor).

By default, 1 in 16 pixels in each 4×4 block of pixels is rendered per frame (see Figure 5.1). The view directions of the non-current pixels are reprojected by the previous frame's view and projection matrices in order to find the pixel in the most closely corresponding pixel in the historic 2D texture.

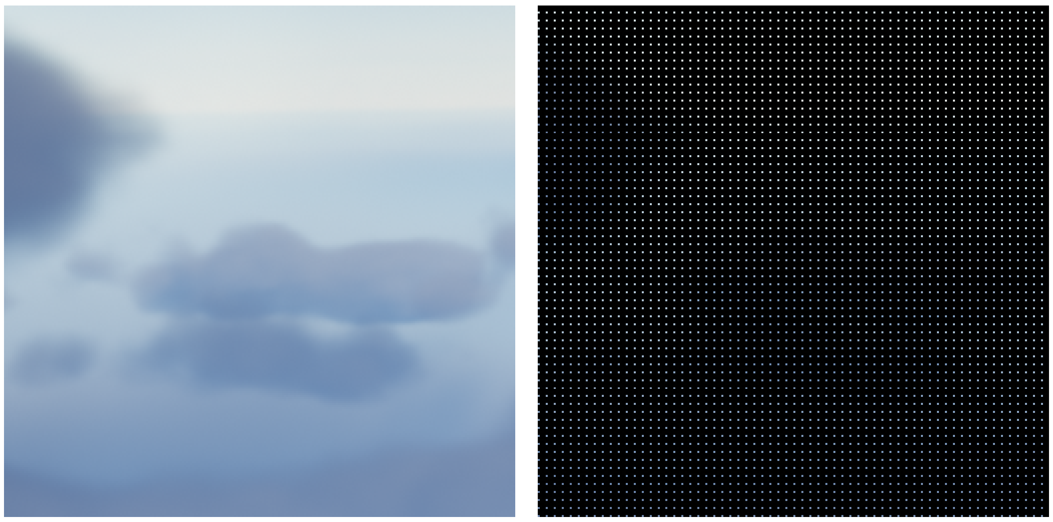


Figure 5.1: A cropped scene rendered fully (left) and the 1/16th of pixels rendered per frame when using reprojection (right).

When the camera is translated or rotated, reprojection may give coordinates outside of the texture; in these cases the closest pixel (in the 2D plane of the rendered image) which was rendered in the current frame will be used instead, to avoid clamping to the edge of the image.

Mini shadow map per node group Ray marching from every voxel towards the light source to compute shadows is extremely expensive. To mitigate the computational cost, rays are first traced from texels on small shadow maps (one map per node group); the per voxel rays then need only ray march to the plane of the group shadow map and sample the group shadow map there.

The gain from this optimisation was around 4-8 times for the shadow computation, varying depending primarily on the highest level of detail in use as well as how low in the sky the light source is. The longer the shadows, the heavier the shadow pass.

Chapter 6
Results

This chapter will present GPU time durations for each compute shader operation, and results of rendering in the form of screenshots and links to videos. All source code created for Mulen is available online at `https://github.com/dat14jpe/mulen` under the MIT licence.

6.1 Hardware and software

Except where otherwise noted, performance results and images were captured on a desktop computer with these defining characteristics:

- Operating system: Windows 10 Pro N, 64-bit
- System memory: 32 GiB
- CPU: AMD Ryzen 7 2700X
- GPU: NVIDIA GeForce GTX 1080 Ti (driver version 445.87)

6.2 Images

Many different cloud views can be produced by the atmosphere renderer, and herein a few sample images are shown and briefly commented.

Mulen has a built-in screenshot feature which encodes the camera parameters into key-value string pairs and encodes them within the output PNG files. Such files can later be dragged-and-dropped onto the program to return to the encoded camera parameters (after waiting a few seconds for the generator to load the target location). This means every screenshot taken within the program also serves as a save point which can be revisited and, for example, re-profiled on different hardware.

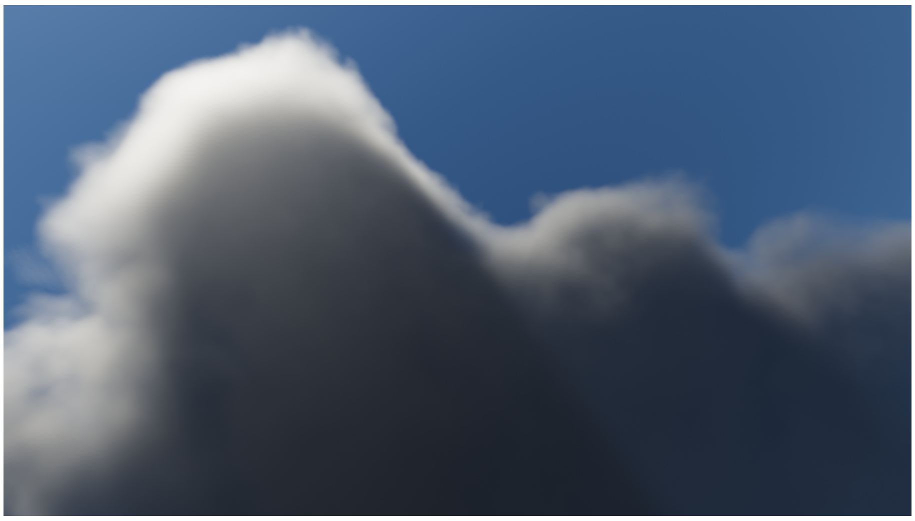


Figure 6.1: An example of silver lining in Mulen.

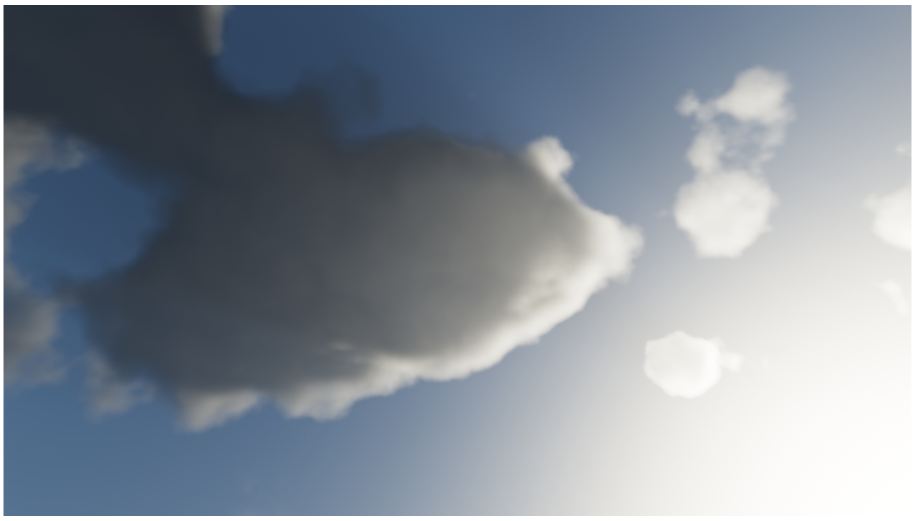


Figure 6.2: Another example of silver lining in Mulen.

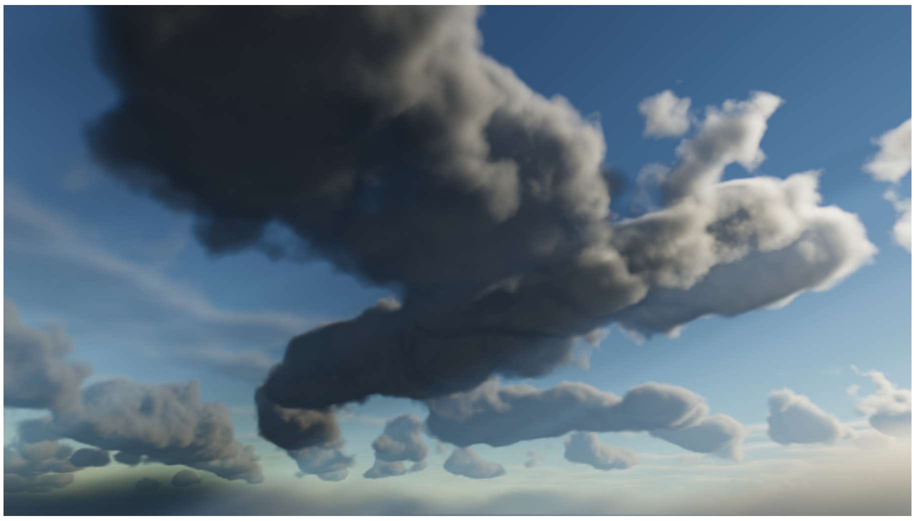


Figure 6.3: Clouds seen from below.

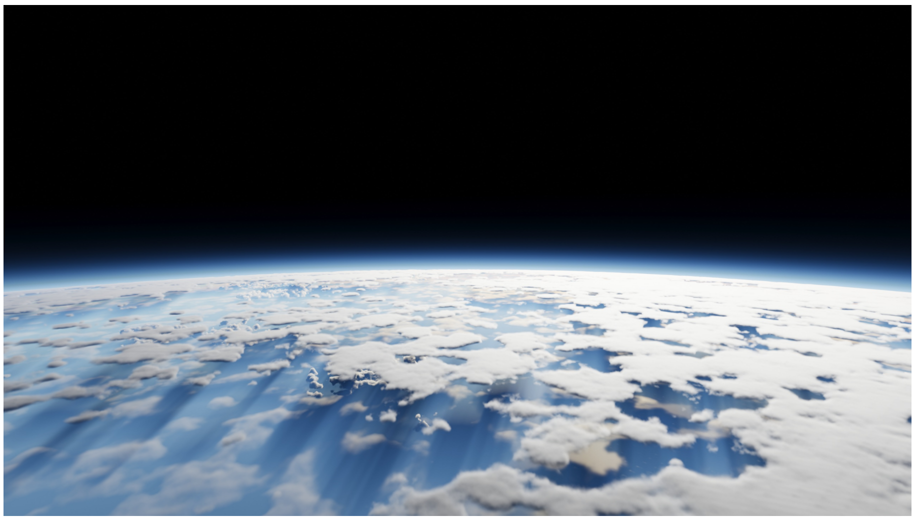


Figure 6.4: A space view including sheet-like altostratus clouds.

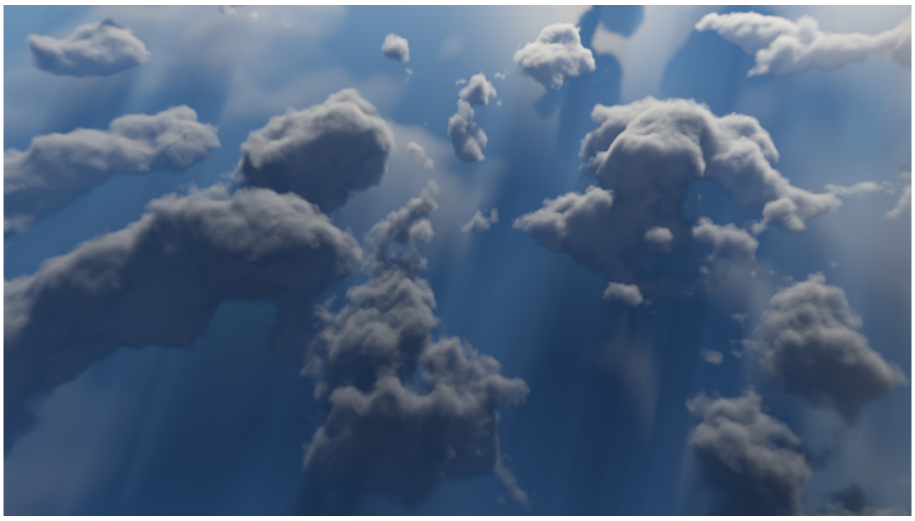


Figure 6.5: Even more clouds (quite seriously).

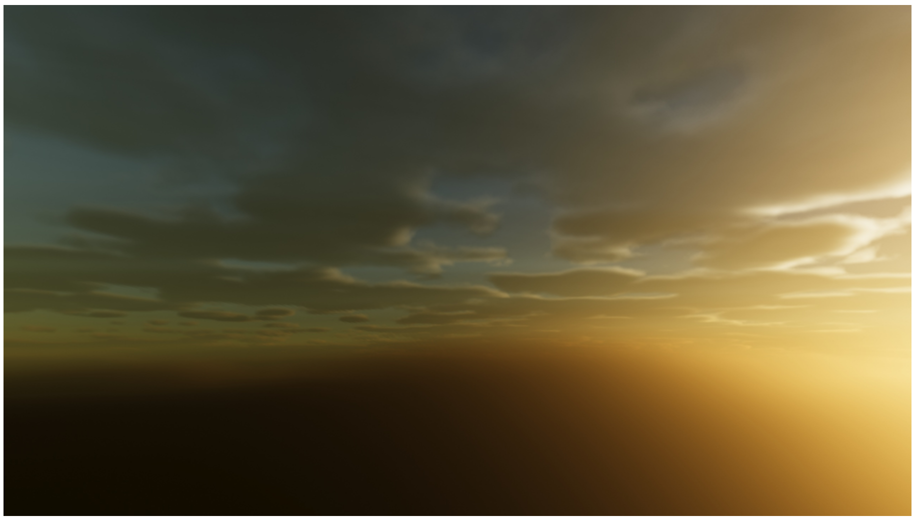


Figure 6.6: A sunset – or sunrise – view showing redder hues.

Mulen does show the silver lining effect due to the strong forward Mie scattering (see Figure 6.1 on page 40 and Figure 6.2 on page 40), can display space views while keeping multiple cloud layers distinct (see Figure 6.4 on the preceding page), and gives the characteristically stronger reddish tones of sunsets and sunrises (see Figure 6.6 on the previous page).

With view frustum culled updates enabled, the entire memory budget is spent on only the part of the atmosphere which is currently directly visible to the camera. This enables extreme levels of zoom while retaining a decent amount of detail (see Figure 6.7), at the cost of needing to wait up to 10 seconds for the updates and generation to adapt fully to any change in camera location, orientation, or field of view.

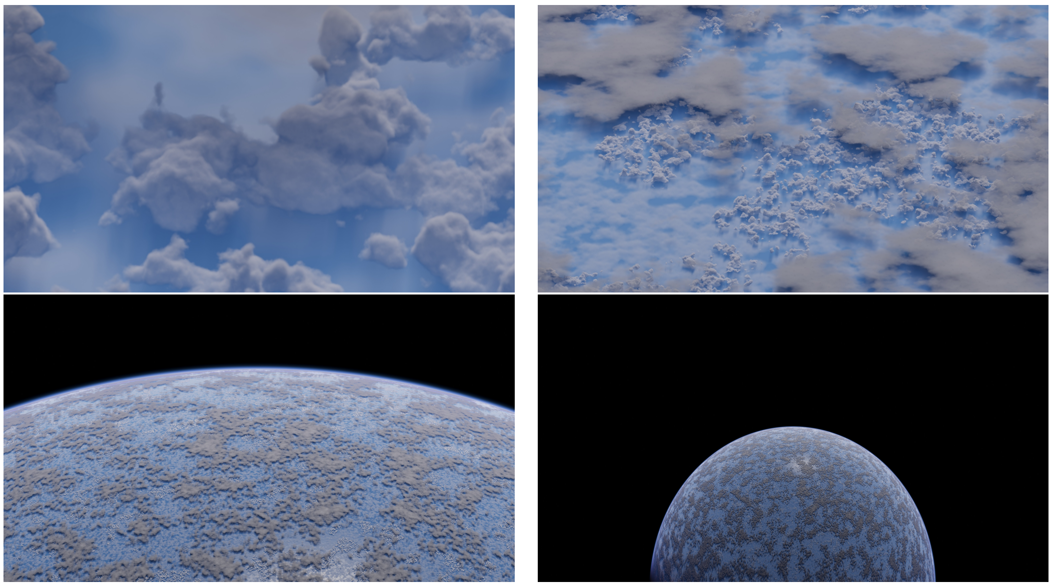


Figure 6.7: Close to 600 times zoom. Field of view (FOV) going from 0.1 degrees (upper left) to 1 degree (upper right), then 10 degrees (bottom left), and finally 58 degrees (bottom right).

The mini shadow map per node group optimisation can introduce undesirable visible boundaries in cloud shadows (see Figure 6.8 on the next page).

6.2.1 Voxel Rayleigh distribution

Early in the course of the thesis work, a separate voxel channel to enable arbitrary Rayleigh density distributions was explored. Low-resolution Rayleigh values and the interpolation across the spherical surface resulted in strong aliasing artefacts (see Figure 6.9 on the facing page). Due to the perceived intractability of solving the problem without increasing required memory amounts dramatically and insignificant visual improvement, the Rayleigh channel was scrapped.

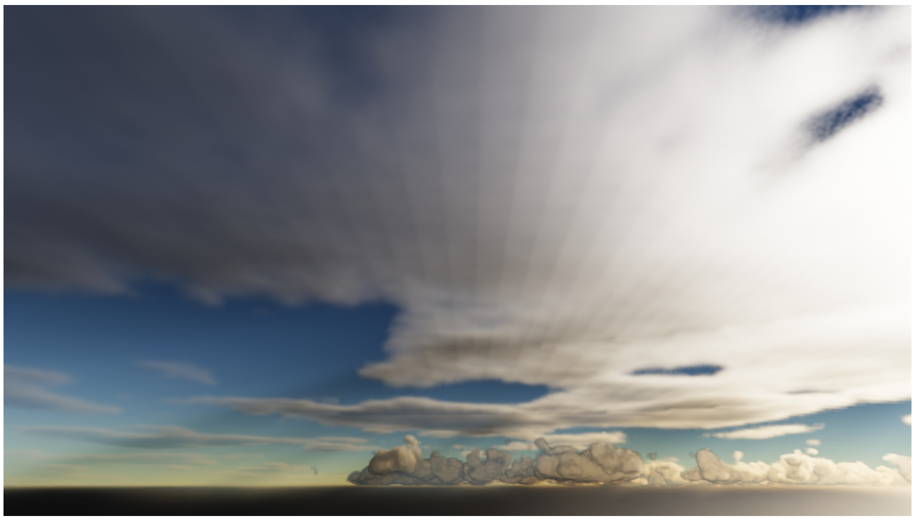


Figure 6.8: Shadow maps per node groups show node group boundaries as lines in the upper clouds.

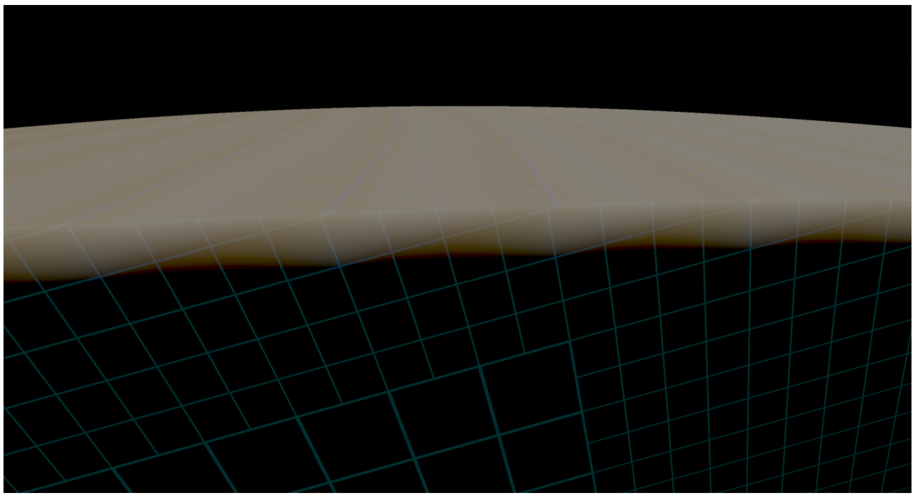


Figure 6.9: A debug view showing a cross section of transmittance due to the voxel Rayleigh distribution, overlaid with grid cell lines. The unevenness in transmittance colour near grid cell corners – i..e, voxel sample locations – shows how interpolated values give rise to highly visible aliasing when sampling the low-resolution voxel grid on a spherical surface.

6.3 Videos

- Animation sample 2020-05-04:
  `https://www.youtube.com/watch?v=pNQPsHTSVX4`
- Animation sample 2020-05-05:
  `https://www.youtube.com/watch?v=prRuflJrqGM`
- Animation sample 2020-05-15:

`https://www.youtube.com/watch?v=OL1wh7T4pGY`

6.4 Performance

All performance values are given as time costs in milliseconds of GPU time. To achieve the desired minimum performance of 30 frames per second, the maximum time cost allowed per frame is $1000\,\mathrm{ms}/30 \approx 33\,\mathrm{ms}$.

The program imposes a high load on the GPU, but only a light one on the CPU (which is not shown here) since most of the update computations are carried out on the GPU. The update thread took around $40\,\mathrm{ms}$ per $1\,\mathrm{s}$ update iteration while running with a GPU memory budget of $2\,\mathrm{GiB}$, varying slightly depending on scene and near-proportional to the memory budget.

The times taken for computing various GPU passes were recorded with the use of OpenGL timestamp queries (retrieved asynchronously one or a few frames after having been recorded, so as not to force a synchronisation point with the CPU which would stall the GPU pipeline) over the course of a few benchmark paths recorded frame-by-frame. All benchmarks were run with vertical synchronisation (V-sync) turned off to avoid more powerful GPUs automatically downclocking and skewing the results by underperforming. Reprojection was set to render only one sixteenth of the full number of pixels per frame.

The CPU time costs observed on the development machine were low: with around $40\,\mathrm{ms}$ for one asynchronous update iteration on the updater thread, run once per second, even a CPU less than a twentieth as fast – on a single core – would still be able to keep up at 60 frames per second. Therefore only GPU times were considered when profiling and benchmarking the program more extensively.

6.4.1 Benchmark 1: variable GPU

Starting above the atmosphere, descending into it, and finally moving at high speed near ground level. Animation interpolation was on. The results displayed here are for 4K resolution (3840×2160 pixels); this is a fairly high resolution, but monitors supporting it are becoming more widespread and thus the performance results at this resolution might remain comparable for near-future comparisons at acceptable quality levels.

The benchmark was run on different computers with various GPUs to give some quantitative sense of how more recent hardware series compare to older ones. Other computer specifics (such as CPUs) were not recorded, since the CPU load is negligible in comparison with the GPU load.

The GPU memory budget was only $1.5\,\mathrm{GiB}$ in order to not exceed the memory amounts in older GPUs (which might have lead to the benchmark not running at all or GPU memory being extended with swapping to and from main system memory, causing a slowdown distorting durations intended to measure computational capacity).

As seen in Figure 6.10 on page 47, the render performance is highly dependent on the view: the camera starts outside the atmosphere, which means rays marched through the atmosphere are short (due to the great distances involved leading to all loaded bricks and voxels being physically big). As the camera moves into the atmosphere, the rays take longer to exit the atmosphere and more computational capacity must be devoted to integrating high numbers of voxels. When the camera speed is increased close to the end, the updates cannot keep up (at the desired level of detail) and thus fewer voxels have to be integrated.

The sawtooth shape in render performance is caused by the low-frequency update iterations (once-per-second) and the camera movement. As an update iteration is just finished, a relatively high amount of detail is loaded close to the camera, giving the rise in render time; as the camera moves away from the position it was in when the last update was begun, the detail seen decreases. Then a new update iteration is completed, and thus emerges the reocurring pattern.

6.4.2 Benchmark 2: variable memory budget

Starting above the atmosphere and descending into it. Run in the resolution 2560×1440, with animation interpolation on, and over memory budgets 2 GiB, 4 GiB, and 8 GiB. This benchmark was only run on the GTX 1080 Ti and RTX 2080 Ti graphics cards since most lower-end GPUs have too low total memory capacity to run the heavier benchmark passes.

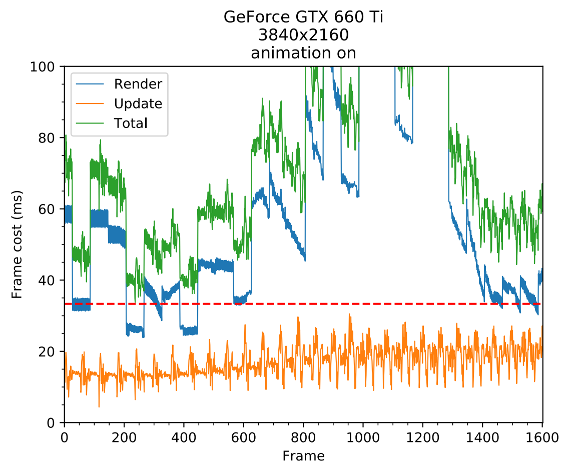
As seen by comparing Figure 6.11 on page 48 with Figure 6.12 on page 49, animation interpolation is very heavy: turning it on slows down the rendering by upwards of a factor of 2.5. Compared with octree-less ray marched clouds taking around roughly 2 ms on somewhat older hardware (e.g. in Häggström 2018 [9], Horizon Zero Dawn [2], and Red Dead Redemption 2 [7]), Mulen is often 8 times slower if given a 4 GiB memory budget (with its total time cost being close to proportional to the memory budget).

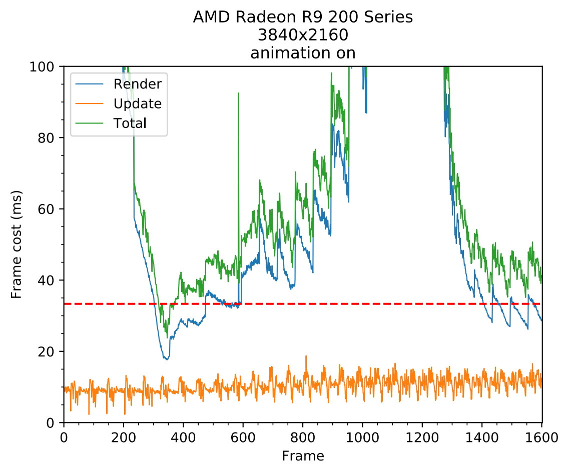
6.5 Image quality

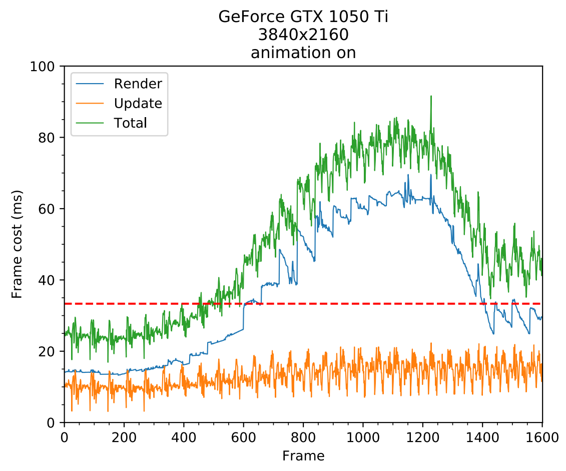
To give a quantitative comparison of image quality, a single view was rendered with 9 different memory budget and step size factor combinations. The highest-quality image (namely the one with the highest memory budget and the lowest step size factor) was used as the reference image, and root mean square error (RMSE) values were computed for each image as compared with the reference image. The maximum octree depth was 13 and the image resolution was 2560×1440.

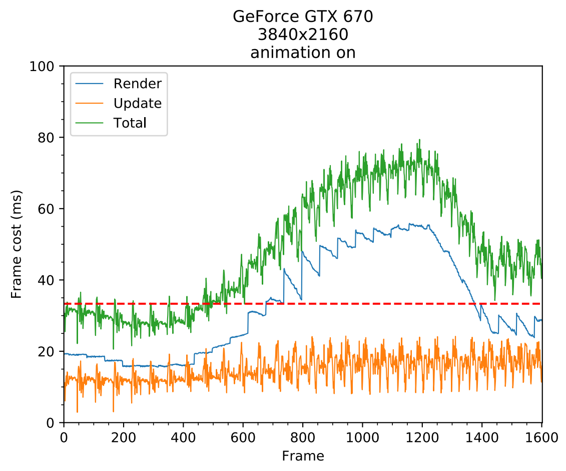
Memory budget/GiB	Step size	Render cost/ms	RMSE	
8	0.1	19.4	0.0	(reference)
8	0.2	12.5	0.5224	
8	0.4	9.9	1.393	
4	0.1	15.2	0.7189	
4	0.2	8.9	0.9706	
4	0.4	6.2	1.755	
2	0.1	12.8	1.105	
2	0.2	7.4	1.374	
2	0.4	4.9	2.182	

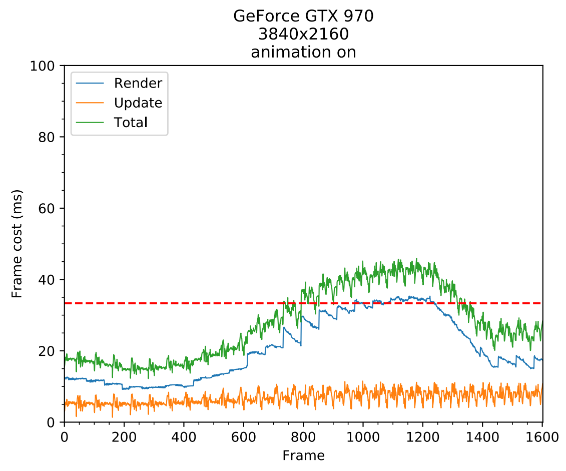
Table 6.1: Render times and RMSE (of pixel RGB triplets on the range $[0, 255]$) for varying memory budgets and step size factors.

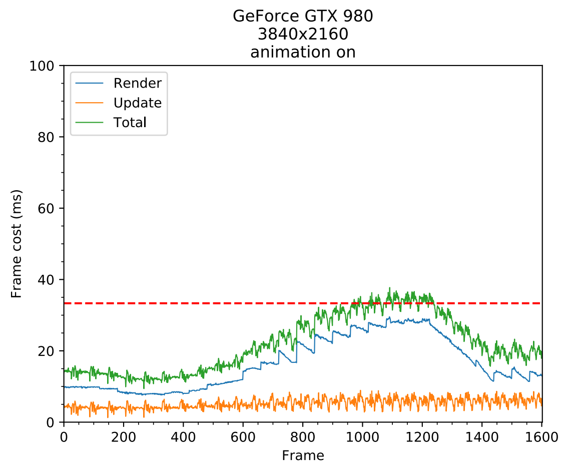

GeForce GTX 660 Ti
3840x2160
animation on
Render
Update
Total
Frame cost (ms)
Frame


AMD Radeon R9 200 Series
3840x2160
animation on
Render
Update
Total
Frame cost (ms)
Frame


GeForce GTX 1050 Ti
3840x2160
animation on
Render
Update
Total
Frame cost (ms)
Frame


GeForce GTX 670
3840x2160
animation on
Render
Update
Total
Frame cost (ms)
Frame


GeForce GTX 970
3840x2160
animation on
Render
Update
Total
Frame cost (ms)
Frame


GeForce GTX 980
3840x2160
animation on
Render
Update
Total
Frame cost (ms)
Frame

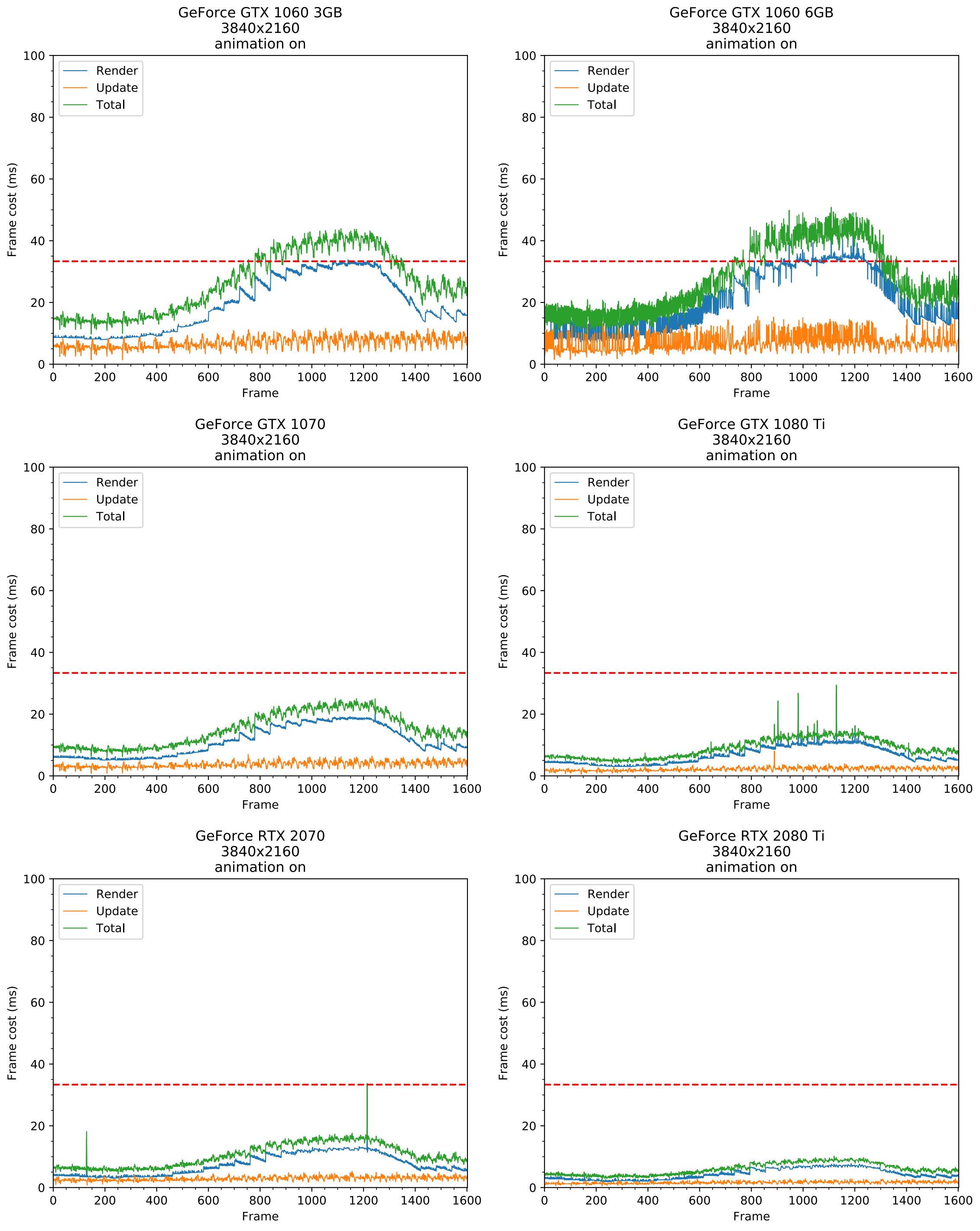


Figure 6.10: Performance results for benchmark 1 (see Section 6.4.1 on page 44). The GPU memory budget was **1536 MiB** and the maximum allowed octree depth was 12. The red dashed line shows the total frametime at 30 FPS.

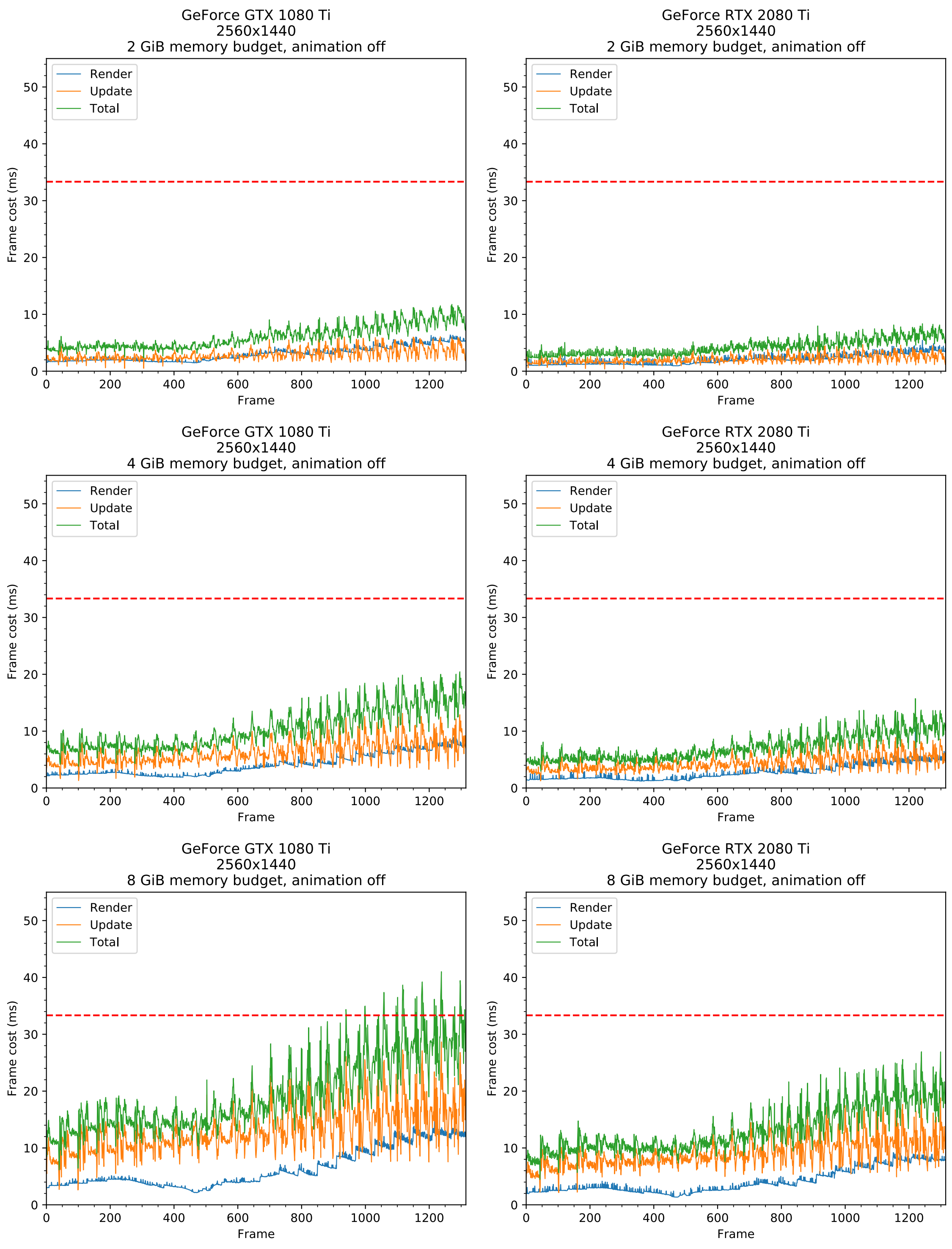


Figure 6.11: Performance results for benchmark 2 (see Section 6.4.2 on page 45) with animation turned off, comparing GTX 1080 Ti with RTX 2080 Ti. The red dashed line shows the total frametime at 30 FPS.

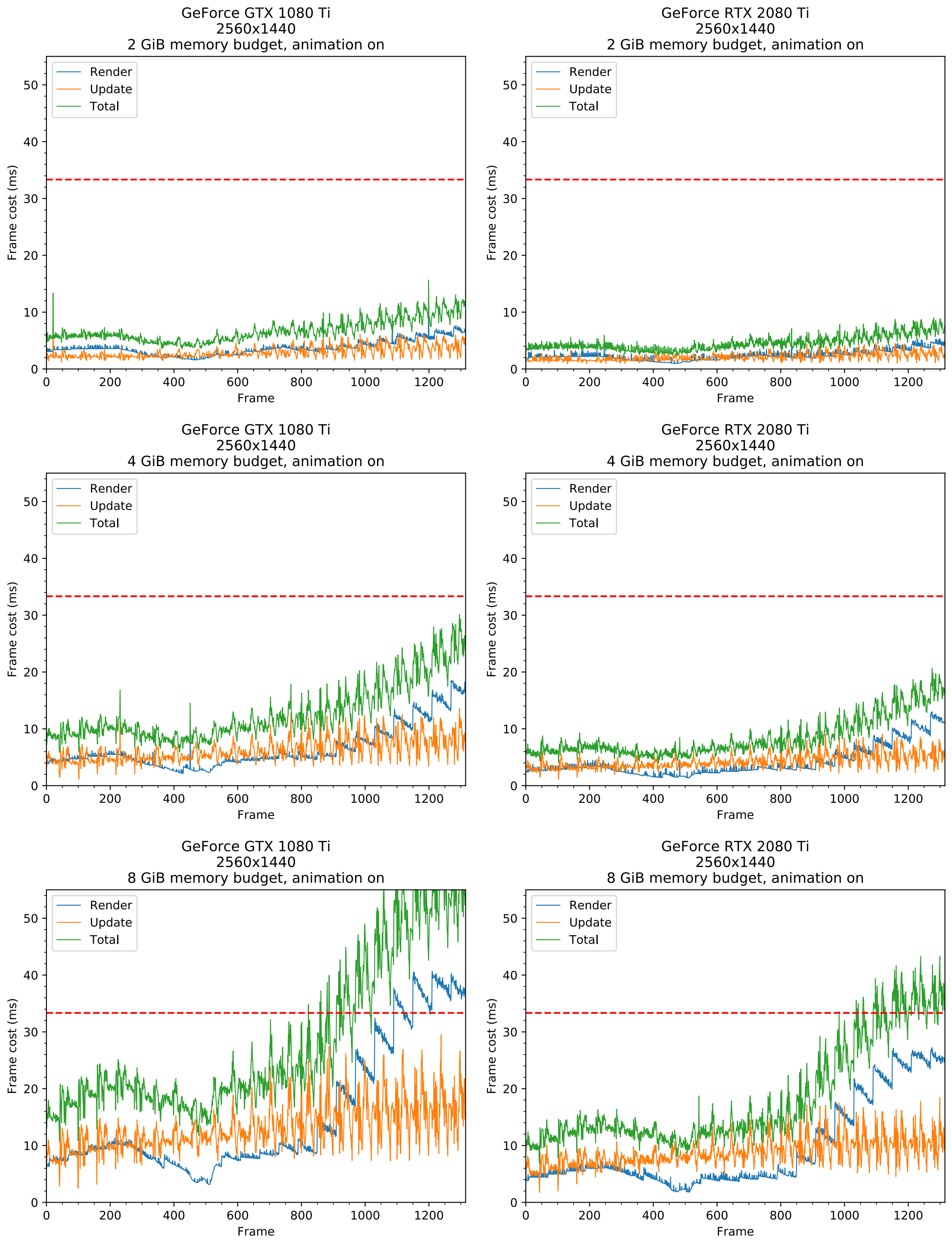
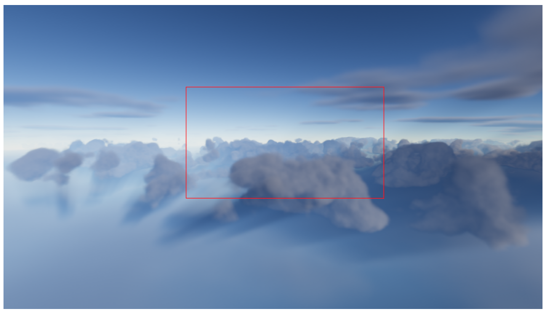
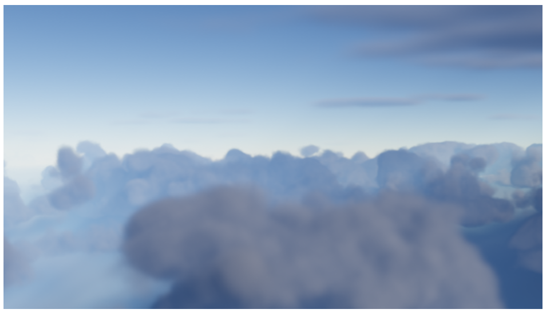


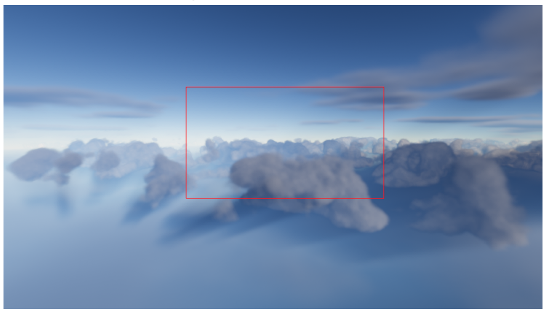
Figure 6.12: Performance results for benchmark 2 (see Section 6.4.2 on page 45) with animation turned on, comparing GTX 1080 Ti with RTX 2080 Ti. The red dashed line shows the total frametime at 30 FPS.

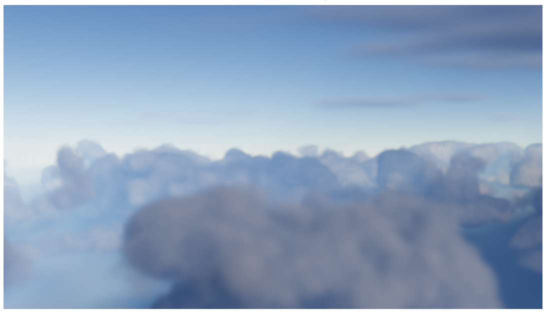
As can be seen in Table 6.1 on page 45, a high step size or low memory budget resulted in lower quality but better performance (i.e., lower render cost). Render time cost increased as the step size was lowered. The step size factor used elsewhere in this work was 0.2, as it was deemed a reasonable compromise between performance and image quality.

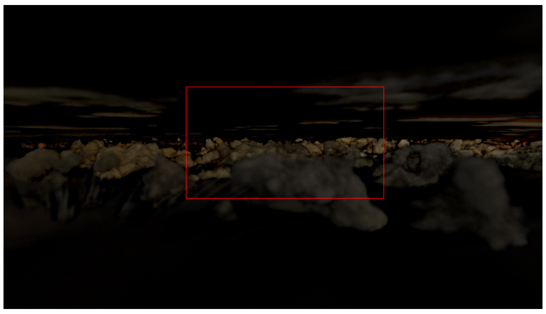
The reference image and the lowest-quality image as well as their per-pixel difference are shown in Figure 6.13, in full and in close-ups of central portions of the full images. Higher step sizes tend to lead to brighter clouds, while a low memory budget causes undesirably diffuse cloud shapes (especially towards the horizon in the shown image).


(a) Reference image (8 **GiB**, step size **0.1**)


(b) Close-up of reference image


(c) Lowest-quality image (2 **GiB**, step size **0.4**))


(d) Close-up of lowest-quality image


(e) Per-pixel difference

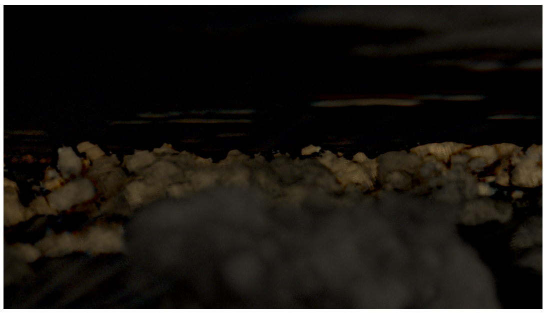

(f) Close-up of per-pixel difference

Figure 6.13: Reference and lowest-quality images as well as their per-pixel difference, in full (left column) and zoomed in (right column). The red rectangle outlines in the full images show the areas which have been zoomed-in to in the images to the right.

Chapter 7
Discussion

The chapter discusses the shortcomings of the work and how they might be adressed, as well as potential future directions of research which might be built on top of the method.

7.1 Shortcomings

Generation aliasing A simple generation shader can easily induce obvious incorrectly darkened lines due to aliasing for example relatively sharp thresholding functions. Care should be taken to avoid features close to or below the voxel size at whichever level of detail is being generated. One way to avoid the problem might be use of distance functions: a scaled and clamped distance field could produce less generation aliasing than a smooth step function defined over distance.

Voxel precision The wide range of cloud densities possible in reality are not easily represented with just 8 bits per voxel density value. 16-bit values might suffice for most common cloud types, but the cost of higher memory bandwidth could be steep for this volumetric rendering. Especially low-density clouds would become possible with a higher bit depth – in particular, noctilucent clouds, which are so thin as to only be visible a period after sunset and were only classified late.

Animation Emulating animation by interpolating between voxel states is both very computationally costly (decreasing render performance by up to approximately a factor of 2.5 in some views, as stated in Section 6.4.2 on page 45) and visually not quite the same as actual movement. Furthermore, the interpolation necessitates keeping an additional set of octree and brick data; if this need could be eliminated – possibly by updating subsets of each brick in smaller update iterations, rather than updating all voxel data in one update iteration – the memory footprint for a given quality level could be reduced significantly.

Movement and update frequency Updating the octree only once per second means the camera cannot move extremely fast without low-detail areas of the octree coming into view. At speeds lower than approximately the escape velocity of Earth (i.e. $40\,000\,\mathrm{km\,h^{-1}}$), low-detail areas were mostly avoided. Visual "popping" from nodes being split to higher detail could be seen occasionally, though the blurring side effect of reprojection smoothed some transitions.

7.2 Future work

Hardware ray tracing for octree traversal At the time of writing (May 2020) some of the most recent GPUs feature hardware acceleration for ray tracing. Use of this hardware feature was considered for Mulen, but the idea was discarded early since Mulen benefits greatly from early exits from shader invocations as the ray transmittance reaches zero in denser clouds, and because atmosphere lighting is non-associative; the samples along viewing rays need to be evaluated in order, which means using the hardware ray tracing would require some sort of per-pixel sorting step as well. I believed the potential performance increases in octree traversal would not offset the performance decrease from needing to temporarily store and then depth-sort often hundreds of sample locations or node indices per pixel.

Higher-order light scattering Only zeroth and first order light scattering were fully accounted for in this thesis. As seen in for example the left half of Figure 6.6 on page 41, locations within the atmosphere that are shadowed from the light source will be overly dark, in contrast with reality where not just one but multiple scattering events may scatter incoming light and soften the distribution of light.

Higher-order light scattering within the clouds was crudely approximated with a function of cloud density, but this only applies to Mie scattering and not to Rayleigh scattering. For a more realistic rendering, higher orders of light scattering need to be approximated more comprehensively.

The cost of integrating second order light scattering could be reduced by only computing it for the corners of the octree nodes and interpolating between them, which would require 1/64th the number of computations compared to carrying them out for every voxel individually. Even so, such an approach would have to integrate over all directions, which would likely require many samples and thus be computationally costly even if carried out only for node corners.

Cone tracing sun with non-zero angular diameter Perfectly thin rays are only accurate if the light source is a point light source, which is not normally the case in realistic outdoor scenes. The sun has an apparent angular diameter, which would need to be considered to create appropriately softer shadows farther from the shadowcasters. Sampling not on a straight ray towards the sun but rather within the cone defined by the ray origin, sun direction, and sun angular diameter could achieve this, though at the cost of needing a full octree descension on every step since reusing the traversal state for samples jittered within the cone would require too branching logic to seem worth the expense in this case.

Multiple light sources If more than one primary light source are required simultaneously, the lighting prepass and lighting textures would need to be duplicated per-light, which would also increase the number of texture samples required for each render ray march step; this is a major weakness of this technique.

So long as a single light source is enough at a single point in time (such as for moonlight and sunlight, as moonlight is only dominant when direct sunlight is not present, i.e. during the night) the lighting prepass could switch light source once per update. This would incur no performance cost.

Advanced generation A generation pass separate from the rendering is both a distinguishing feature and significant disadvantage for Mulen. In principle, this allows for maximum flexibility in cloud shapes; but in practice, it also requires very large amounts of memory and computational resources. When running on current real hardware, this poses challenges.

Alternative data sources Since generation and rendering are separated in Mulen, and rendering only makes use of values computed by generation, it is possible to change the generation method without necessitating any difference in the render code. A more sophisticated procedural generator could be substituted, perhaps cloud values scanned from reality could be streamed, or the results of some sort of fluid simulation could be displayed. The architecture of the program makes this possible without delving into the render parts.

Extraterrestrial atmospheres The Rayleigh, Mie, and ozone constants could be changed to approximate another atmosphere than Earth's, as could the radius of the atmosphere and the light intensity value. It is also possible to create another cloud generator routine specialised for different cloud formations.

Chapter 8
Conclusion

In conclusion, the sparse voxel octree atmosphere implementation developed as part of this thesis did succeed in being able to display views ranging from outer space and all the way down to the clouds and ground level below them. However, it was found to be highly computationally demanding and just slightly less highly demanding in memory use, even while missing a way to integrate higher-order light scattering. I consider it an interesting idea, but ultimately not nearly as efficient as would be needed for most applications where the atmosphere isn't the sole load on the graphics card (and only possibly sufficiently efficient for those applications where it is). Only with significant performance improvements could it be made practically useful.

References

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*, page 1200. A K Peters/CRC Press, Boca Raton, FL, USA, 2018. pp. 11-27, 29-30, 54.

[2] Andrew Schneider. SIGGRAPH 2017: Advances in Real-Time Rendering, Nubis: Authoring Real-Time Volumetric Cloudscapes with the Decima Engine, 2017. `https://www.guerrilla-games.com/read/nubis-authoring-real-time-volumetric-cloudscapes-with-the-decima-engine`, slides at `https://d1z4o56rleaq4j.cloudfront.net/downloads/large/Nubis%20-%20Authoring%20Realtime%20Volumetric%20Cloudscapes%20with%20the%20Decima%20Engine%20-%20Final.pptx`.

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

[4] Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. English and web-optimized version.

[5] Earth Science and Remote Sensing Unit, NASA Johnson Space Center. Thunderstorms over Borneo, astronaut photograph ISS040-E-88891, August 2014. `https://earthobservatory.nasa.gov/images/84308/thunderstorms-over-borneo`. [Accessed: 2020-05-27].

[6] Eric Bruneton. Precomputed atmospheric scattering: a new implementation, 2017. `https://ebruneton.github.io/precomputed_atmospheric_scattering/`.

[7] Fabian Bauer. SIGGRAPH 2019: Advances in Real-Time Rendering, Creating the Atmospheric World of Red Dead Redemption 2, 2019. `https://advances.realtimerendering.com/s2019/index.htm`, slides at `https://advances.realtimerendering.com/s2019/slides_public_release.pptx`.

[8] Stefan Gustavson. Simplex noise demystified, 2005. `http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`.

[9] Fredrik Häggström. Real-time rendering of volumetric clouds, 2018. https://umu.diva-portal.org/smash/record.jsf?pid=diva2%3A1223894&dswid=2065.

[10] Stephen Hill. https://github.com/therealmjp/bakinglab/blob/master/bakinglab/aces.hlsl, 2016. https://github.com/TheRealMJP/BakingLab/blob/master/BakingLab/ACES.hlsl.

[11] Luke Howard. Essay on the Modifications of Clouds. *The Askesian Society, London (1796-1807)*, 1803.

[12] Carlos Jiménez de Parga Bernal Quirós. *High-Perfomance Algorithms for Real-Time GPGPU Volumetric Cloud Rendering from an Enhanced Physical-Math Abstraction Approach.* PhD thesis, UNED. Universidad Nacional de Educación a Distancia (España), 2019.

[13] Kelvinsong. Earth's atmosphere, January 2013. https://commons.wikimedia.org/wiki/File:Earth%27s_atmosphere.svg. [Accessed: 2020-05-29] Licenced under the Creative Commons Attribution-Share Alike 3.0 Unported licence, available at https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[14] Rikard Olajos. Real-time rendering of volumetric clouds, 2016. Student Paper.

[15] Ken Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296, New York, NY, USA, 1985. Association for Computing Machinery.

[16] T Smith and J Guild. The C.I.E. colorimetric standards and their use. *Transactions of the Optical Society*, 33(3):73–134, jan 1931.

[17] https://github.com/Dav1dde. glad: Multi-Language Vulkan/GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs., 2020. https://glad.dav1d.de/.

[18] https://github.com/g-truc. OpenGL Mathematics (GLM), 2020. https://glm.g-truc.net/.

[19] https://github.com/glfw. GLFW: A multi-platform library for OpenGL, OpenGL ES, Vulkan, window and input, 2020. https://www.glfw.org/.

[20] https://github.com/lvandeve. LodePNG: PNG encoder and decoder in C and C++., 2020. https://github.com/lvandeve/lodepng.

[21] https://github.com/nlohmann. JSON for Modern C++, 2020. https://nlohmann.github.io/json/.

[22] https://github.com/ocornut. Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies, 2020. https://github.com/ocornut/imgui.

[23] Valentin de Bruyn / Coton. Cloud types en, January 2012. https://en.wikipedia.org/wiki/File:Cloud_types_en.svg. [Accessed: 2020-05-29] Licenced under the Creative Commons Attribution-Share Alike 3.0 Unported licence, available at https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[24] Patricio Gonzalez Vivo and Jen Lowe. The Book of Shaders: Fractal Brownian Motion, 2015. https://thebookofshaders.com/13/.

[25] WhiteTimberwolf. Octree2, March 2010. https://commons.wikimedia.org/wiki/File:Earth%27s_atmosphere.svg. [Accessed: 2020-05-29] Licenced under the Creative Commons Attribution-Share Alike 3.0 Unported licence, available at https://creativecommons.org/licenses/by-sa/3.0/deed.en.

[26] Wikipedia. Beer-Lambert law. https://en.wikipedia.org/wiki/Beer%E2%80%93Lambert_law.

[27] Wikipedia. Octree. https://en.wikipedia.org/wiki/Octree.

[28] Wikipedia. Sparse voxel octree. https://en.wikipedia.org/wiki/Sparse_voxel_octree.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-12

EXAMENSARBETE Real-time rendering and dynamics of sparse voxel octree clouds
STUDENT Johan Pettersson
HANDLEDARE Pierre Moreau (LTH)
EXAMINATOR Michael Doggett (LTH)

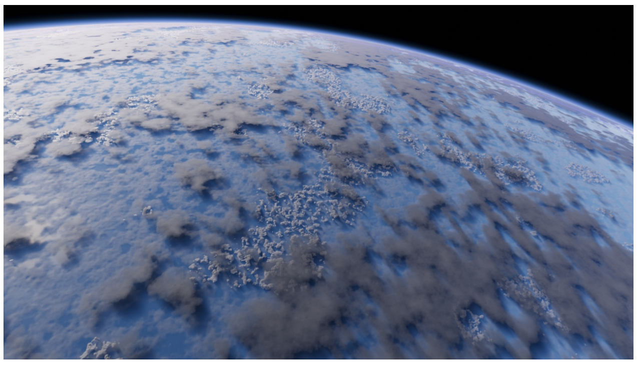
Atmosfärrendering med volymetriska moln och skuggor

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johan Pettersson**

Volymetriska moln är idag vanliga i spelgrafik. Men molnen byggs ofta med upprepade detaljer, vilket kan bli uppenbart bristfälligt när kameran rör sig upp i och över molnskiktet. Detta gör det svårt att hantera perspektiv från både marknivå och höjder ute i rymden och övergångar mellan dessa.

När ljus når atmosfären så fortsätter en andel av ljuset i den riktning det var på väg i, en annan andel absorberas av luften, och den sista andelen sprids i alla riktningar. Spridningen är inte lika fördelad över alla riktningar eller våglängder, vilket ger det starka skenet i dimma och mot solen samt atmosfärens färger. Atmosfärrendering bygger på att numeriskt integrera dessa effekter över de geometriskt beräknade sträckor i atmosfären som ljuset färdas innan det når kameran. Sådana beräkningar kan förberäknas för en klar atmosfär utan moln och sedan effektivt renderas.

I det här arbetet skrevs ett program som använder en spatiell partitioneringsstruktur för att kunna representera molntätheter i en atmosfär. Täthetsvärdena lagras i 3D-block av volymetriska element – så kallade voxlar – som kan sträcka sig över olika fysiska storlekar beroende på hur djupt fördelad strukturen är. En gång varje sekund uppdateras strukturen så att stora voxlar som är långt bort från kamerans nuvarande läge kan förkastas för att ge minnesplats åt mindre och närmare voxlar. Tack vare detta kan programmet exekvera inom en förbestämd total minnesmängd och ändå visa förhållandevis små molnformationer när kameran befinner sig nära ytan.



I bilden ovan visas en renderad vy över en jordstor planet. Molnformationer sträcker sig till den synliga horisonten och molnens skuggning av atmosfären visar tydligt ljuskällans riktning.

Avslutningsvis uppnåddes realtidsprestanda (det vill säga över 30 bildrutor per sekund) med vissa lägre kvalitéinställningar, fast detta utan hänsyn tagen till högre ordningars ljusspridning. Tidigare realtidsmetoder är betydligt snabbare men programmet skrivet under detta arbetes gång kan hantera unika molnformationer över avsevärt större skalor, vilket är en ovanlig egenskap för molnrendering i realtid.