APPROVAL SHEET

Title of Thesis: GPU-accelerated Rendering of Atmospheric Glories

Name of Candidate: Ari Rapkin Blenkhorn
Doctor of Philosophy, 2018

Thesis and Abstract Approved: _____

Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

Title of Thesis: GPU-accelerated Rendering of Atmospheric Glories

Ari Rapkin Blenkhorn, Doctor of Philosophy, 2018

Thesis directed by: Marc Olano, Associate Professor
Dept. of Computer Science and Electrical Engineering

Incorporating atmospheric phenomena such as glories into games and other interactive graphics applications increases the visual realism of natural environments in those applications, creating a more immersive and believable virtual world. This work presents techniques for rendering glories quickly and accurately for use in such applications.

The glory appears as a collection of concentric colored rings, akin to the larger, better-known rainbow. Its color banding pattern is described by the Mie scattering equations, which are complex and must be calculated for many different wavelengths and scattering angles. This dissertation presents a novel implementation of Mie scattering which performs these calculations in parallel on the GPU using OpenGL compute shaders. It achieves significant rendering speedups over previous sequential CPU implementations of glory rendering. Additional algorithmic refinements are supported by the radial symmetry of the glory and the limited range of physical scenarios in which glories occur.

The number of Mie calculations required can be substantially reduced without sacrificing perceptual accuracy by selecting Mie scattering inputs using 2D low-discrepancy Sobol sampling in (wavelength, scattering angle) space rather than independent 1D wavelength selection per pixel. The contributions of this work include the GPU implementations of several Sobol variants and findings on their relative benefits.

An incremental rendering framework spreads the Mie scattering calculations over multiple frames to achieve interactive speeds. It begins with a fast approximation render which is gradually refined. It uses performance prediction to respond to a changing time budget and assesses render status and image quality using multiple metrics.

Performance comparisons are provided for the Mie scattering shaders on various hardware configurations.

GPU-accelerated Rendering of Atmospheric Glories

by

Ari Rapkin Blenkhorn

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

This work is dedicated to the memory of Professor Randy Pausch,

my teacher, mentor, and inspiration.

You taught me the value of brick walls and Tiggers.

Sorry about the snake bite.

Acknowledgments

Thanks to my advisor, Marc Olano, for recognizing my strengths and accepting my weaknesses with respect and humor. He found the perfect balance and made my return to grad school not only a rewarding challenge, but also a heck of a lot of fun.

Thanks to Penny Rheingans for being one of my dissertation readers. Penny was the first person at UMBC I approached about returning to school. Her informative, welcoming response made a world of difference.

For many helpful comments, thank you to my other reader, Adam Bargteil; and to the rest of my committee: Curtis Menyuk, Matthias Gobbert, and Raymond Lee.

Theory and initial exploration of the Reordered Sobol method were the work of Marc Olano and his colleagues during his sabbatical at Epic Games. Thanks to the Epic group for suggesting partial-index table-lookup Sobol as well.

Solar spectral irradiance data from SORCE SIM was obtained from the Laboratory for Atmospheric and Space Physics, University of Colorado, Boulder. Thanks to Jerry Harder for the data and the good wishes.

A few more thank-yous to companions who shared this journey:

My husband Kevin and my children Dottie and Marc made this adventure possible with their patience, enthusiasm, and understanding.

Sally McKee and Lauren Bricker have been my friends, colleagues, and guiding stars for over twenty years, through two rounds of graduate school and everything between.

Wes Griffin, Yu Wang, and Mark Bolstad, the graphics Ph.D. students who preceded me at UMBC, were a source of camaraderie and guidance.

Tucker and Glory, my dissertation kitties, offered reassurance and affection while making sure I didn't leave my desk.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A great deal of recent research in computer graphics has focused on generating realistic images, which accurately depict physical phenomena such as sunlight on water, mirror reflections, and the translucent glow of skin. However, physical realism usually comes with a high price tag, as the algorithms are computationally expensive, the simulations must be high-resolution, and the offline preparation to run them is significant.

In many computer graphics applications, rendering involves solving the integral of one or more complicated functions. Often the analytical solutions to these integrals are unavailable, or cannot be solved efficiently. To provide rapid, high-quality solutions, we often use numerical or probabilistic methods. The well-known Monte Carlo method approximates the desired function by running the calculation a large number of times, using input values randomly selected from the allowed range. Because true random sequences are unrepeatable and may display clumps and gaps of arbitrary size, quasirandom sequences are often used instead. These offer many of the same features as random sequences while also ensuring that samples are evenly distributed throughout the sample domain. Quasirandom sequences are generated in a deterministic manner, enabling independent components of a larger system to share the same sequence without explicit coordination. An extension of Monte Carlo, called Importance Sampling, also samples points in a similar manner, but not evenly. Guided by an approximation of the function being computed, it samples more frequently at more "important" regions of the domain. This technique can converge on the integral more quickly than pure Monte Carlo if the approximating function is well-chosen.

As computer performance has increased, so have the demands on this capability. Rendering and simulation algorithms have increased in complexity in the constant drive toward

improved realism. For example, 20 years ago, cloth simulations were simple spring-and-mass meshes (Baraff and Witkin 1998). Newer features include self-collision, dynamic tearing, plastic deformation, interaction with water, and numerous other features that bring the simulation ever closer to a faithful reproduction of actual cloth (see Rapkin (2002), Rapkin et al. (2003), Bridson, Fedkiw, and Anderson (2005), and Fei et al. (2018) for examples).

For a physicist studying atmospheric phenomena, the goal is to produce a point-by-point accurate representation of the spectral irradiance values. For computer graphics, this level of physical realism can be excessive and inappropriate. It is focused on the wrong metric, as our purpose is to produce an equivalent visual experience for human observers, as viewed on physical devices. Our measures of image accuracy must consider the capabilities and limitations of the human visual system and the displays we use.

The purpose of my dissertation is to improve rendering of the colorful atmospheric phenomena which occur through the interaction of sunlight and cloud droplets. The system presented here can do this 1) rapidly and accurately enough for inclusion into games and other interactive applications and 2) within the limited computation and memory resources generally available on consumer systems. Additionally, the improvements I have made in the speed and accuracy of these techniques are useful to physicists and other scientists who may make a different speed-vs-resolution or speed-vs-accuracy tradeoff yet still appreciate a significant speedup over their previous methods.

1.1 Scope, limitations, and assumptions

I have established optimized GPU-based rendering methods for glories. These methods are also applicable to coronas and rainbows, as these three phenomena are closely related. All three are created by light interacting with clouds or mist. They appear as collections of concentric colored rings, though in the case of the rainbow typically the full rings are not visible and only the upper part (a “bow”) is seen. The glory, being the smallest, is almost always visible as a complete circle. These three phenomena span as much as three orders of

magnitude of cloud droplet size, and occur across the full hemisphere of sun-cloud-viewer geometries from the corona, which surrounds the sun at a radius of a few degrees, to the rainbow at 42° outside the antisolar point, to the glory, which closely surrounds the antisolar point.

I have targeted upper-tier consumer-level hardware with single GPU configurations, as the primary application for this work is for dedicated but recreational game players who can be assumed to invest in good machines but not go to extremes. However, as this research is also intended to contribute speed and convenience for high-precision, large-scale research into the physics of glories and related phenomena, I have included timing results for key components on more powerful systems to show how they perform given exceptional resources.

I have not developed new mathematical formulations for any of the optics and atmospheric physics in this work. The mathematical descriptions of Mie scattering and associated phenomena have been well-explored over more than a century, and serial, CPU-based algorithms have been created based on the established equations (Bohren and Huffman 1983). In this dissertation I present parallel, pure-GPU implementations which take advantage of the novel strengths and capabilities of today's graphics hardware. This work focuses on producing perceptually-accurate images for use in interactive applications, and with that goal in mind, leverages characteristics of the human visual system to increase effectiveness.

1.2 Significance

Incorporating atmospheric phenomena into games and other interactive graphics applications will increase the visual realism of natural environments in those applications. Greater visual realism enables a more immersive and believable virtual world. In a scientific setting, it also provides more physical insight.

Very little research exists on optimizing the rendering of this entire family of optical phenomena on modern graphics hardware. The closest example is the work done in ren-

dering glories, coronas, and fogbows on the CPU (Section 2.5). The techniques developed for optimization in the context of video games will likely also be applicable and desirable in support of ongoing research into the optics of these phenomena. While physicists may require higher resolution or more physical accuracy than can be achieved in real-time, the overall computation times are still significantly faster. Furthermore, the results of my work will enable physicists to generate artificial images of atmospheric phenomena using fewer resources than before, without sacrificing confidence in the physical accuracy of the result.

Finally, the field of optical physics is concerned with producing images as a tool to study the the behavior of light whereas computer graphics is concerned with the production of images for human view. These communities assess images based on different criteria. Most work in atmospheric phenomena has been done by the physical science communities, and is not well-suited for graphics applications. Perceptually-based objective metrics for quality of rendering of these phenomena provide the graphics community with a rigorous metric for objective evaluation of rendered atmospheric phenomena as they relate to the human visual experience.

1.3 Contributions

A key contribution of this work is the OpenGL compute shaders which perform the Mie scattering calculations. These code components perform the same calculations as their linear CPU algorithms, but because the GPU can compute many instances of the algorithm at once, realistic scenarios requiring thousands of Mie calculations complete in far less time.

A second contribution is the conceptual reformulation of the glory rendering task as a 2D sampling problem of selecting (wavelength, scattering angle[1]) pairs, rather than a 1D sampling problem of selecting wavelengths for each pixel. Using samples which are evenly distributed across this combined domain produces comparable results to the 1D method with far fewer Mie calculations. Alternatively, higher-quality results can be produced with

[1]The angle by which the water droplet deflects the incoming light ray.

the same number of Mie calculations.

A third contribution is the OpenGL compute shaders which perform the table-based 2D Sobol sampling and cell-indexed Sobol sequence generation, and my findings on the relative benefits of different implementation methods and lookup table structures.

1.4 Organization of the next several chapters

In Chapter 2, I present a variety of key concepts, information, and technologies which the reader may not be familiar with, as these establish the foundation on which the research contributions of subsequent chapters are constructed.

Chapter 3 presents the structure and capabilities of ComputeGlory, the rendering and testing framework application developed for this research. It provides details of the programming environment, computing platform, and auxiliary data used for the application. This chapter also outlines the quality metrics and timing mechanisms used to capture the data reported in subsequent chapters.

Chapter 4 quantifies the improvements obtained by the application of modern graphics and parallel computing techniques to update the serial CPU implementation of BHMIE. I describe my implementations of two variants of the BHMIE scattering algorithm as OpenGL shaders, along with an approximation function derived from best-available Mie scattering results, and the circumstances under which each of these three shaders is used. I also discuss the physical parameters of Mie scattering which are exposed to user control, how those parameters relate to rendering properties and visual characteristics of the output images.

Chapters 5 through 7 address the conceptual / core contributions of this work. These chapters compare the performance characteristics of the final glory rendering system against the baseline system quantified in Chapters 4.

Chapter 5 takes a deeper look at the sampling methods used in ComputeGlory and gives details of their implementations and relative performance.

Chapter 6 investigates two methods of selecting the wavelengths and scattering angles which are presented as input to the Mie scattering shaders.

Chapter 7 looks at the tradeoffs of distributing the Mie calculations over multiple frames. Several options are explored, including various end criteria, adaptive cohort sizing, and rendering a simpler function to generate a fast approximation which is refined with complete Mie results over successive frames.

Chapter 2

BACKGROUND

This chapter presents an overview of several background topics which the reader will need to understand, as they form the foundation of the research contributions presented in subsequent chapters.

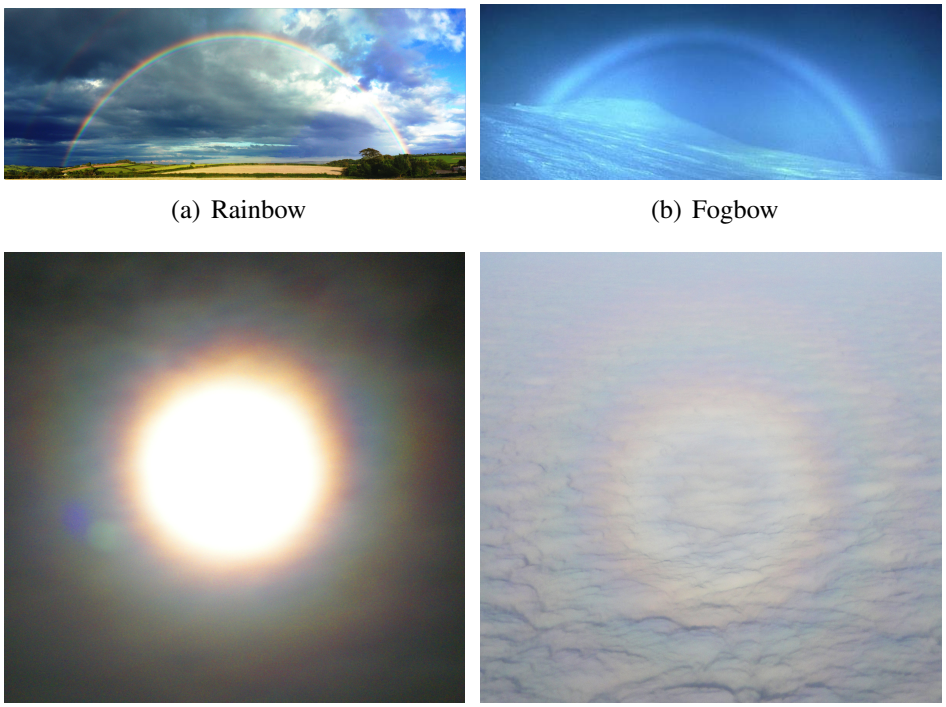
- Atmospheric glories and other wavelength-dependent atmospheric optical phenomena
- Mie scattering theory and computational methods
- Color spaces and perceptual color metrics
- Color rendering of atmospheric phenomena
- Real-time rendering
- OpenGL
- General-purpose GPU computing
- Monte-Carlo methods and importance sampling
- Sampling with random, pseudorandom, and quasirandom sequences

2.1 Atmospheric glories

Various colorful optical effects are visible in the sky when the meteorological conditions and the geometry of light, clouds, and observer are cooperative. These phenomena are caused by the reflection, refraction, and diffraction of light with small water droplets or ice crystals in the atmosphere. In this work, I am only considering effects seen when sunlight interacts with

liquid water in the earth's atmosphere. Furthermore, I am only considering scenarios where the droplets of water are all the same size, that size lies within a certain commonly-observed size range, and the droplets are spherical. This section presents examples, describes their physical characteristics, and introduces techniques for simulating these phenomena, which are discussed further in Section 2.2.

2.1.1 Observational characteristics

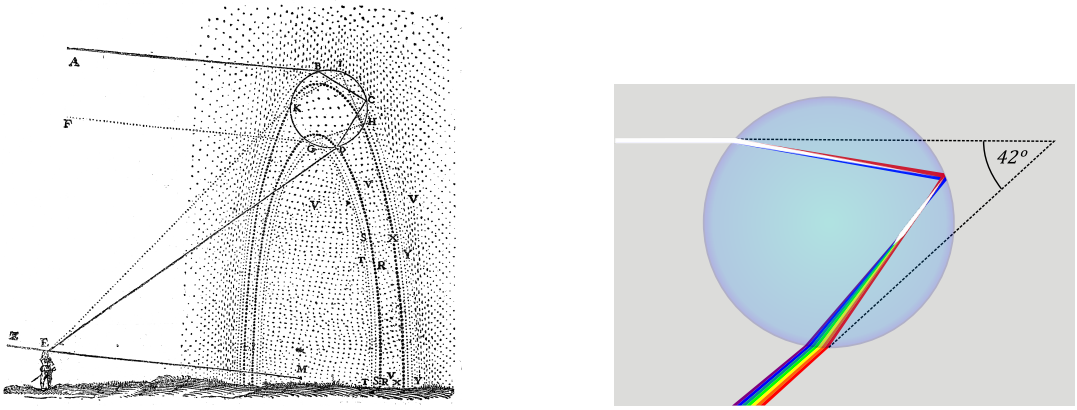


(a) Rainbow (b) Fogbow

(c) Lunar corona (d) Glory on clouds

Figure 2.1: Examples of atmospheric phenomena. For image source information, see Appendix B.

The best-known such effect is the rainbow (Figure 2.1(a)), the multi-colored arch seen when the air is full of large droplets, usually during or just after a rainstorm. It is possible to see the complete circle of a rainbow, but this is unusual due to the geometry required. Rainbows are caused by reflection and refraction of sunlight coming from behind the observer, and form a band at approximately 42˚ from the antisolar point, containing one repeat of the spectrum color sequence. (Figure 2.2 (a))



(a) Sun, observer, antisolar point, and rainbow (b) Reflection and refraction within a droplet

Figure 2.2: Formation of a rainbow. For image source information, see Appendix B.

Closely related is the cloudbow or fogbow (Figure 2.1(b)). These are the same phenomenon, called by different names to indicate where and how it is observed. They occur on droplets 10 to 100 times smaller than rainbows, so that the size of light waves becomes significant. Diffraction causes the colors to blend together into a white arch.

The corona (Figure 2.1(c)) is most often seen at night, in mist around the moon. It displays several concentric colored rings around a white center and is caused by diffraction.

The glory (Figure 2.1(d)) resembles a small circular rainbow and is frequently seen from aircraft when the observer is above a cloud layer and looks directly opposite the sun. Glories are also sometimes seen by skydivers looking down through thin cloud layers, or by people looking down from high bridges into mist over churning water. They are always centered around the shadow of the observer's head or camera.

Through observation (Laven 2006) and experiments with artificial clouds (Ray 1923), glory rings and coronas have been determined to have very different distributions of brightness and colors.

Which optical phenomenon is visible depends on the size of the droplets in the air. Most glories are caused by droplets between 4 and 25 μm. The color bands' widths vary inversely with the droplet size. Although the color bands are also affected by the refractive index of the droplets, there is very little visible difference caused by the variation in re-

fractive indices encountered in the water droplets composing earthly clouds (Laven 2008a), while droplet size has a strong effect. Coronas are generally seen at $10 - 40\mu$m, fogbows are typically seen when droplets are about 50μm, and rainbows 300μm or more, with the clearest above 1mm. The size parameters (ratio of droplet circumference to wavelength) of rainbows are so large that the color banding can be explained through geometric optics and Airy scattering as well as through Mie scattering (discussed below). Airy theory has clear inaccuracies when applied to small droplets, but for a realistic drop-size distribution of large droplets, it is quite close to Mie theory in perceptual metrics such as chromaticity and luminance contrast (Lee 1998).

All these phenomena are most vivid with monodisperse droplets (all one size), which is unusual in natural clouds; or when the droplet size distribution is very narrow (Laven 2005).

Supernumerary rainbows are pale, purplish bows which sometimes are visible directly inside the primary rainbow. These cannot be explained by geometric optics. They are caused by interference between two sections of the same light wavefront as it traverses a raindrop, reflects, and folds over on itself.

2.1.2 Simulation theory and implementation

For some of the atmospheric phenomena discussed here, the physics are well understood. For example, the familiar rainbow is caused by light rays that enter a water droplet, reflect once internally, and exit, with refraction at entrance and exit spreading out the constituent wavelengths. (See Figure 2.2 (b).) Other phenomena are less clearly understood. The physics behind the glory has been explored and debated for over a century (Schröder and Wiederkehr 2000; Ray 1923; H. Nussenzveig 2002).

Fortunately, for purposes of rendering these phenomena, it is not necessary to explicitly simulate the physics of the interaction between the light and the water. We need only a mathematical description of the light patterns that result: the wavelengths, exit angles, and

intensities that result when light with a known spectrum enters. These are well-established and understood. The next section presents further details.

2.2 Mie scattering

Gustav Mie's solutions (Mie 1908) to Maxwell's equations (Maxwell 1873) provide an exact description of the way a monochromatic electromagnetic plane wave is scattered by a homogeneous sphere. Variants include scattering by coated spheres, ellipsoids, cylinders, and other types of particles. Generally, the term "Mie scattering" is used for scenarios where the size of the particles is comparable to the wavelength of the light, as compared to Rayleigh scattering (Strutt 1871) for small particles up to about 10% of the incident wavelength and Rayleigh-Gans-Debye (RGD) scattering (Debye 1944) for large particles and colloidal suspensions. RGD scattering places restrictions on both particle size and refractive index. However, Mie scattering has no such limitations. Also, Mie scattering has a much larger forward-scattering component than Rayleigh scattering.

Given the sphere size, the complex-valued refractive index of the sphere, and the wavelength of incident light, the Mie scattering equations give the outgoing intensity at a specified angle. When calculated over a span of wavelengths, the results feature particularly strong or weak bands at certain scattering angles, known as the Mie resonances.

The Mie scattering intensity is represented as the sum of an infinite series of terms representing the contributions of partial waves. Even though in practice only a finite number contribute much to the sum, Mie's method still requires evaluating a large collection of complicated mathematical expressions (H. M. Nussenzveig 2012). The calculation includes determination of scattering phase function along with extinction, scattering, and absorption efficiencies. Several other methods of computing scattering have been developed which provide good approximations under certain conditions. These include the Debye series expansion (Debye 1908), Airy theory (Airy 1838), modified geometric optics (discussed in Greenler (1980)), and T-matrix generalization to non-spherical particles (Waterman 1965).

2.2.1 Debye series

The Mie equations are often rewritten in a different form. The Debye series (Debye 1908) decomposes the partial wave scattering amplitudes a_n and b_n into an infinite series of contributions corresponding to diffraction, external reflection, and transmission following 0 or more internal reflections of the partial waves. The usual notation to describe these cases denotes the partial wavenumber as n and the number of internal reflections as $p - 1$ where $p = 0, 1, 2, \ldots$The case of direct transmission with no internal reflections is simply $p = 1$. $p = 0$ describes the case where no transmission occurs at all; i.e. external reflection.

The Debye expansion is not an approximation to Mie theory. The summation of the Debye series for all integer values of p from zero to infinity equals the Mie result. It is a different technique for arriving at the same result, or a means for approximating it efficiently. Shen and Wang (2010) present an improved method for calculating the Debye series. However, they acknowledge that calculating the Debye series expansion remains much more difficult than using the Mie formulation, and is not the preferred method unless the partial wave decomposition is needed.

2.2.2 History of Mie scattering in practice

It is generally agreed (e.g. Greenler (1980), Bohren and Huffman (1983), and Laven (2008b)) that Mie theory leads to good simulations of atmospheric phenomena but does not explain the actual physical processes happening in and around the water droplets. Given this lack, and given how computationally demanding Mie theory can be for the scattering of a continuous spectrum (600+ wavelengths needed for glory with radius $r = 10\mu m$), it was not met with widespread adoption for quite some time after its initial introduction by Gustav Mie in 1908.

In 1958, van de Hulst published lookup tables, bringing Mie theory to attention as possessing practical value. The first computer implementation was Dave's DBMIE (1968). Subsequently, several improvements to the original Mie scattering algorithm have been

developed. Warren J. Wiscombe (1979) developed *a priori* criteria for determining from x (the size parameter, defined as $x = \frac{2\pi r}{\lambda}$ where r is the radius of the droplet and λ the wavelength of the incident light) and m (the complex index of refraction) whether upward recurrence can be used safely in a particular Mie calculation or whether downward recurrence is necessary. Upward recurrence is much faster, and thus preferred whenever it is safe to use. Wiscombe also restructured some of the recurrence equations to make them easier and faster to use. He presented other improvements to the existing Mie scattering calculations for very small particles, and for vector-structured computation. The BHMIE algorithm (Bohren and Huffman 1983) provided a more efficient method than DBMIE to determine the convergence of the series, and incorporated Wiscombe's NSTOP criterion, recognizing that the loops could be terminated after a fixed number of iterations related to the size parameter of the droplets.

Mie theory was finally used to simulate atmospheric effects in color in the late 1990s (Lee 1998; Brandt and Greenler 2001; Gedzelman and Lock 2003). Over the next few years, complete simulation applications were developed such as MiePlot (Laven 2003) and IRIS (Cowley 1998; Cowley, Laven, and Vollmer 2005).

MiePlot (Laven 2003) is written in Visual Basic and was originally meant as a simple interface to the classic BHMIE code. The current application, available for download from Laven's website (http://www.philiplaven.com/mieplot.htm), provides graphs of intensity vs. scattering angle, wavelength, radius and refractive index. MiePlot calculates scattering for a single wavelength or a sunlight distribution. It offers a breakdown into Debye series components, enabling consideration of the contributions due to individual ray paths. MiePlot also features Airy theory, Rayleigh scattering, diffraction, and ray tracing including interference between rays.

IRIS (Cowley 1998; Cowley, Laven, and Vollmer 2005), available for download from Cowley's website (http://www.atoptics.co.uk/droplets/iris.htm), simulates coronas, glories, and fogbows by computing Mie scattering intensities over a range of angles and hundreds

of wavelengths in the visible range. The results are combined in a weighted sum using the spectral power distribution of incident sunlight. IRIS also supports either monodisperse droplets or size distributions.

Brewer (2004) showed that pre-computed rainbows and fogbows can be used as textures in order to give a real-time effect. However, the optics calculations were done offline using MiePlot.

Cowley and Schroeder built an ice-crystal simulator, HaloSim3, which creates simulations of halos and arcs resulting from light rays interacting with ice crystals. Its Monte Carlo ray tracing engine uses geometric descriptions of ice crystals and their orientations to the sun and horizon, then traces up to several million rays as they enter, reflect, and refract through crystals. HaloSim3 is available for download at
http://www.atoptics.co.uk/halo/halfeat.htm.

Cowley and Schroeder modified HaloSim to create BowSim, which uses ray tracing to simulate rainbows from large raindrops. Where HaloSim traces light rays through ice crystals, BowSim traces them through spheres and general ellipsoids. BowSim is available for download at http://www.atoptics.co.uk/rainbows/bowsim.htm.

Cowley and Schroeder also developed AirySim, which uses Airy theory to simulate white-light rainbows. Results are comparable to, but much faster than, Mie theory. It can simulate non-monodisperse droplets by repeating the basic Airy calculations for each droplet radius in its droplet population. AirySim is available for download at
http://www.atoptics.co.uk/rainbows/airysim.htm.

Riley et al. (2004) used the MiePlot system to calculate scattering to 0.1˚ degree accuracy, which is insufficient for high-resolution graphics applications. The glory typically is visible to about 5˚ out from the antisolar point , so for a 512x512 image we need to perform calculations to approximately 10˚/512 or 0.02˚ resolution.

Sadeghi et al. (2012) simulate the activity of light rays within large, non-spherical droplets to yield atypical rainbow phenomena such as twinned bows. I have limited my work

to spherical- droplet rainbows in order to support rendering multiple phenomena within a unified framework, and glories and coronas are the product of small, spherical droplets. In fact, the non-spherical nature of the droplets studied by Sadeghi et al. (2012) is directly attributed to their large size, so this method does not apply to the smaller-droplet phenomena.

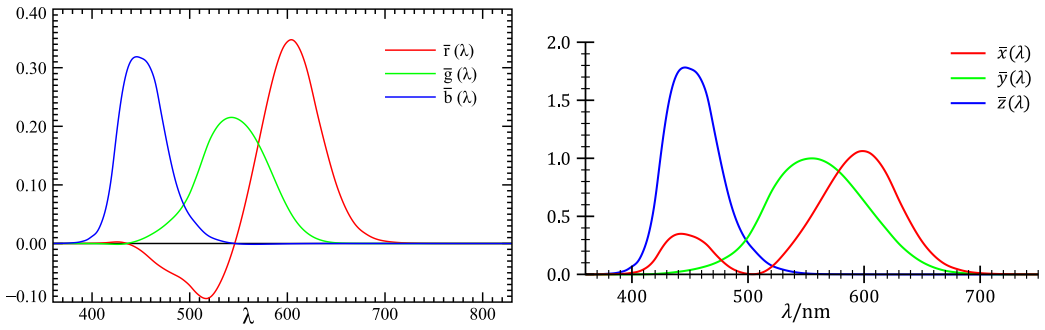
Furthermore, their work is a physics-based method which directly represents the geometry and behavior of light rays as they enter, interact with, and exit the droplet. This type of simulation of glories and coronas is not necessary for rendering, when the Mie scattering equations fully describe the output light intensity. For glories, it would be impossible, because as previously mentioned, the physical basis of the glory is not completely known.

2.3 CIE color functions

The CIE 1931 RGB and XYZ color spaces (CIE 1931) describe the quantitative relationship between pure physical stimuli (spectral radiances) and the colors that are perceived by human vision. This mapping serves as a standard reference and enables translation between human vision and different display and recording devices such as inks, monitors, and digital cameras. It includes all colors that human vision can perceive.

The corresponding color-matching functions enable us to perform calculations which represent light as a collection of wavelengths and intensities, then translate to perceptual color values. Because the human eye has three kinds of color-sensitive cone cells, with sensitivity peaks in three different regions of the visible spectrum, we can parameterize any color by three values, corresponding to the level of stimulation experienced by each cone type.

The CIE 1931 RGB color functions shown in Figure 2.3(a) were determined by an experiment in which participants were shown a test color and asked to reproduce it as a combination (possibly negative) of three primary colors at 700nm (red), 546.1nm (green) and 435.8nm(blue). The aggregated results are known as the CIE 1931 RGB Color Matching Functions or *standard observer functions*. Denoted as $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$, they give



(a) RGB (b) XYZ

Figure 2.3: CIE 1931 color matching functions for RGB and XYZ spaces. For image source information, see Appendix B.

the tristimulus XYZ values for a wavelength λ. These values are available in tabular form at 1nm intervals for the spectrum from 380nm to 780nm at http://cvrl.ioo.ucl.ac.uk/cmfs.htm or in Wyszecki and Stiles (1982). Results from a similar experiment in 1964 are available but less often used. The latter may actually be the better fit for graphics and wide FOVs, as the 1964 data used a 10˚ sample versus the 1931 experiment which used a 2˚ sample.

Rather than using sampled values, analytical fit functions to $\bar{x}$, $\bar{y}$, and $\bar{z}$ are preferred, so that the XYZ conversion can be applied at any wavelength. However, the tabulated data is not amenable to fitting with simple curves. Initial fitting attempts used computationally expensive functions but lacked computing support for an extensive exploration of the parameter space, so results were poor. Wyman, Sloan, and Shirley (2013) offer fits for both the 1931 and 1964 observers. These functions are not intended to fit perfectly, only within the measured variance between subjects. They provide two fit functions: a single-lobe analytic fit for both 1931 and 1964 data, and for 1931 only, a more accurate multi-lobe piecewise-Gaussian fit. I have used the 1931 (2˚) multi-lobe fit to convert from wavelength intensity to XYZ values.

The CIE XYZ color functions (Figure 2.3(b)) were constructed from the RGB functions to meet several requirements including that the transform from RGB to XYZ is linear, and that the values of X, Y, and Z are always non-negative.

The RGB-to-XYZ matrix is specified in the CIE standards exactly.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.1}$$

The XYZ-to-RGB transform uses the inverse, which is not specified in the standards but can be derived:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.1847 & -0.15866 & -0.082835 \\ -0.091169 & 0.25243 & 0.015708 \\ 0.00092090 & -0.0025498 & 0.17860 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.2}$$

However, the RGB color space of the 1931 CIE experiment is not the only such color space in common use. I have used sRGB, which is a widely used color space created for monitors, printers, and the internet. It is designed to match typical illumination conditions for homes and offices, and is defined based on CIE Standard Illuminant D65, which roughly corresponds to midday sunlight. Like the original RGB-XYZ conversions above, to convert from CIE XYZ values to displayable sRGB requires only a matrix multiply.

2.4 L*a*b* color space and CIE 1994 ΔE^* color distance function

A color distance function gives a method for calculating a number that represents the distance between two colors. A color distance of 1.0 is intended to be the minimum that the human eye can discern (often abbreviated "JND" for Just Noticeable Difference). Colors closer than 1.0 are perceived as the same; colors farther apart are perceptibly different.

However, because the human eye is not equally sensitive to all wavelengths, defining a suitable function has not been a simple task. In the original formulation ($\Delta E76$), distances were defined as Euclidean distances in the CIE L*a*b* color space (CIE 1976). This color space was intended to be "perceptually uniform", meaning that equivalent color distances

in different parts of the color space should have equivalent perceptual differences. Unfortunately, this was not the case, especially in saturated colors.

Given two colors (L_1^*, a_1^*, b_1^*) and (L_2^*, a_2^*, b_2^*) the difference between them is:

$$\Delta E_{76}^* = \sqrt{(\Delta L)^2 + (\Delta a)^2 + (\Delta b)^2} \tag{2.3}$$

where

$$\Delta L^* = L_1^* - L_2^*$$

$$\Delta a^* = a_1^* - a_2^*$$

$$\Delta b^* = b_1^* - b_2^*$$

Several other variations have been devised since 1976, with improvements to perceptual uniformity and reference data (Robertson 1990).

The ΔE_{94}^* color distance function (Commission Internationale de L'eclairage 1995) is designed to provide a perceptually uniform distance metric. Given two pairs of colors (A, B) and (C, D), if A and B (the colors in one pair) appear to a viewer to be as far apart as C and D (the colors in the other pair), then their distance as indicated by ΔE_{94}^* are the same. This holds throughout the color space, which is itself non-uniform to account for the human eye being more sensitive to some colors than others.

More recent formulations exist, but they are far more complex. ΔE_{94}^* is acceptable for just color differences (Fairchild 2005).

$$\Delta E_{94}^* = \sqrt{\left(\frac{\Delta L^*}{k_L S_L}\right)^2 + \left(\frac{\Delta C_{ab}^*}{k_C S_C}\right)^2 + \left(\frac{\Delta H_{ab}^*}{k_H S_H}\right)^2} \tag{2.4}$$

where

$$C_1^* = \sqrt{a_1^{*2} + b_1^{*2}}$$
$$C_2^* = \sqrt{a_2^{*2} + b_2^{*2}}$$
$$\Delta C_{ab}^* = C_1^* - C_2^*$$
$$\Delta H_{ab}^* = \sqrt{\Delta a^{*2} + \Delta b^{*2} - \Delta C_{ab}^{*\,2}}$$
$$S_L = 1$$
$$S_C = 1 + K_1 C_1^*$$
$$S_H = 1 + K_2 C_1^*$$

k_C and k_H are both usually 1 and the weighting factors $k_L = 1.0$, $K_1 = 0.045$, and $K_2 = 0.015$ are indicated as appropriate for graphics applications in the ΔE_{94}^* definition (Commission Internationale de L'eclairage 1995).

2.5 Sources of error in color rendering of atmospheric phenomena

Our goal is to produce on a computer screen an image which a human observer considers as indistinguishable from what a human sees when looking at a glory in the sky. This process must take into account the complexities of the human visual system, and differences in displayed color caused by differences in the characteristics of display systems (e.g. desk monitor, mobile screen, projector) and display environments (e.g. indoor office lighting, outdoor sunshine, outdoor cloudy day). To produce perceptually-accurate results, the rendering system must address the correspondence between computed wavelength and intensity, computed RGB, displayed (device) RGB, and the colors perceived by the human observer. (Laven 2003, 2005)

Color renderings require the scattering intensities for each wavelength to be accumulated and the result converted into displayable RGB triples. For each value of θ (scattering angle), the final output indicates the corresponding brightness and color of light. To represent this information compactly, Lee (1998) introduced what came to be known as Lee

Diagrams: rectangular images, essentially a matrix where each entry indicates the color of light scattered in a specific direction (θ) by a drop of radius r. (See Figure 2.4.)

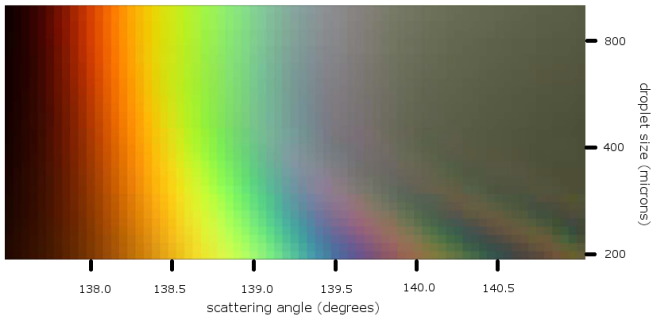


Figure 2.4: Lee Diagram, modified from Lee (1998) with permission

Laven (2003) gives a list of potential sources of error:

- **Spectral radiance of sunlight**. Laven uses measurements from Lee (1998).
- **Refractive index of water as function of wavelength**. There is ongoing disagreement on this matter. Laven uses values published by The International Association for the Properties of Water and Steam (1997).
- **Color conversion to RGB**. Laven uses Bruton (1996), which resembles the CIE XYZ mapping discussed in Section 2.3 but uses piecewise linear functions.
- **Color characteristics of display device**. Intrinsic characteristics such as gamma and device color gamut affect the colors perceived by the user, as do user settings like contrast, brightness, and color temperature.
- **Failure to account for effects of indirect illumination**. Although this work addresses phenomena which result from single scattering off droplets at the front face of the cloud, multiple scattering off the deeper cloud contributes to the perceived appearance.

2.6 Real-time rendering

Rendering is concerned with synthesizing images with a computer. *Real-time rendering* refers to the collection of activities which together make these images *rapidly*. New images must be generated and presented quickly enough that the user can interact with the system, with no perceived delay between user actions and the appearance of images that are generated in response to those actions. This is particularly important in 3D immersive environments, where delay can lead not only to annoyance but to actual physical distress (Kolasinski 1995; LaViola 2000).

The exact definition of real-time depends on context. Usually, rendering speed is measured in frames per second (fps) or milliseconds per frame. The standard display rate for projected film is 24 fps (42ms/frame), and for video and television, 25 or 30 fps is common (40 or 33ms/frame, for PAL/SECAM and NTSC). Multiples of these frame rates are sometimes seen, with correspondingly smaller times per frame. Although these are display rates for previously-created imagery, they provide useful comparisons as they demonstrate the frame rates that users frequently encounter and come to expect in interactive viewing as well.

Computer games typically run at 30 fps (33ms/frame), with some recent games targeting twice that rate, 60 fps (16ms/frame). Virtual reality, however, requires an even higher refresh rate for user comfort. High-end VR headsets such as the HTC Vive (HTC Corporation 2018) and Oculus Rift (Facebook Technologies, LLC 2018) currently feature a 90 fps refresh (11ms), and if the scene is fully rendered twice per frame (once for each eye), the computation time is further reduced.

These are total time budgets for the entire rendering process. A particular component of the rendering system, such as Mie scattering, may be allocated a certain number of milliseconds in which to complete all calculations and render its contribution.

Consumer desire for real-time rendering of complex scenes continues to increase, as viewers expect interactive applications to present images of equivalent quality to the most

cutting-edge cinematic special effects. To feed this hunger, consumer graphics hardware has been growing rapidly over the last couple of decades. Where all rendering calculation was once done by the CPU, now graphics processing units (GPUs) are common in desktop and laptop computers.

2.7 OpenGL

OpenGL (Open Graphics Library, http://www.opengl.org) is a widely used API for rendering graphics. Originally released in 1992, today it is implemented by most graphics cards and it has been used to create thousands of applications on systems ranging from supercomputers to mobile devices. These applications include broadcasting, mechanical design, games, films, medical imaging, scientific visualization, and virtual reality. Over time OpenGL has expanded to include many new features. Most relevant for my work are the OpenGL Shading Language (GLSL), which permits programmer control of several stages of the rendering pipeline, general-purpose compute shaders, and the extensive support for texture usage.

Changes to the OpenGL API are well-controlled, publicized, and supported; and backward compatibility requirements support developers who cannot or will not upgrade to each new version. OpenGL is consistent, in that the visual results of an OpenGL application are equivalent on any system that supports OpenGL. At the same time, it is evolving, incorporating new developments rapidly through an extension mechanism which offers them as optional features. This collection of features makes it appropriate for a long-term project.

2.8 General-purpose GPU computing

GPUs were created to accelerate graphics processing, but they are not limited to graphics. They are highly-capable parallel processing devices which can be used for non-graphics data-parallel computation as well. This is general-purpose GPU computing, or GPGPU. Even a single GPU-CPU system has advantages over a multi-CPU system because the GPU

and CPU are each specialized for certain types of computing. GPUs are designed to accelerate performance of computationally-intensive floating-point operations with efficiency that the general-use CPU cannot match. Many universities and research labs including UMBC support GPU clusters where many powerful GPUs are linked to perform particularly large or rapid computation. For example, as of February 2018, the UMBC High Performance Computing Facility (https://hpcf.umbc.edu/) features a total of 324 compute nodes in a variety of configurations, including 19 CPU/GPU nodes, each offering two NVIDIA K20 (NVIDIA 2012) GPUs. The K20 has 2496 CUDA (compute) cores and 5GB of onboard memory, and the recently released NVIDIA Volta GV100 GPU has 5120 CUDA cores and 16GB memory.

For comparison, the work presented in the subsequent chapters was performed on a 2014 mid-level gaming laptop, with a single NVIDIA GeForce GT 750M (Kepler architecture) GPU. This GPU has 384 cores and 2GB memory (NVIDIA 2018b).

Additional comparisons between the GT 750M and the GV100 are provided in Appendix A.

Software developers have several options for how to interact with general-purpose GPUs. Initially, in the early 2000s, the only way to communicate with the GPU was through a graphics API. General-purpose problems had to be recast to appear to be graphics operations. Though effective, this was convoluted and difficult to debug. Today, we have tools intended for this use. Some APIs, such as NVIDIA's CUDA, have abstracted away any relationship to image processing and do not require any graphics programming experience at all. The recent trend of GPUs being utilized for bitcoin mining illustrates their desirability for non-graphics applications (Mearian 2018).

OpenGL *compute shaders* provide a GPGPU framework within OpenGL. They are a comparatively recent extension to core OpenGL with version 4.3 in 2012. The compute shader model supports resource-sharing and synchronization between the compute and render aspects of an application. The host (CPU) application mediates data ownership transfer

between the host program and compute shaders.

The name *compute shaders* reflects their heritage. Early graphics shaders were concerned only with shading calculations in the graphics pipeline. Today all GPU computational programs are known as shaders, and distinguished by secondary identifiers which indicate their function: fragment shaders, vertex shaders, and so on.

Compute shaders are not automatically executed by the OpenGL rendering pipeline. The host program calls OpenGL functions to dispatch groups of shader invocations, which share user-defined parameters, textures, and shaderbuffers, but are differentiated by a built-in input which gives each invocation's index within the group. The index identifies the invocation, enabling it to determine what data to process and where to store it. The group sizes are three-dimensional, but any of the dimensions can be limited to a single value, making the overall computation effectively lower-dimension. For example, to work pixel by pixel within a 512x512 image, specifying group sizes as (512, 512, 1) yields a 2D index whose dimensions correspond to the pixel coordinates.

A single application can employ multiple compute shaders with different invocation counts, group sizes, and parallelism schemes. For example, one shader can perform computations for each sample in a 1D dataset, then another shader can use that dataset to render each pixel of a 2D image.

Shader Storage Buffer Objects, or shaderbuffers, were introduced along with compute shaders. They provide a mechanism to store and retrieve large, variable-size arrays of read-/write data between compute shaders and their host program. Although buffer objects already existed in OpenGL, the earlier variants had much lower size limits along with other restrictions which made them unsuitable for general purpose compute shader usage. The host program calls OpenGL's `glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT)` synchronization functions between compute shaders to ensure that earlier shaders have completed all operations affecting shaderbuffers before the next shader begins.

Control is passed back and forth between CPU and GPU as each shader stage com-

pletes, but data remains on the GPU unless explicitly requested by the CPU host program. As a result, the time-consuming transfer of data between CPU and GPU can be avoided. This is an important consideration. Modern GPUs supply massive parallel processing power, but often they are not fully utilized because moving data between GPU and CPU is such a bottleneck. To get the most out of the GPU, it's important to minimize data transfer and keep the GPU busy.

2.9 Monte Carlo methods and importance sampling

Monte Carlo methods are methods of computing the integral of a function over a region when either the integral or the region are not amenable to analytical methods. The rendering equation in physically-based rendering is an example, as realistic lighting scenarios often yield complicated functions which are not readily integrated. To perform Monte Carlo integration, a number of points are sampled from the domain, the value of the function is computed at those points, and the results are averaged.

Monte Carlo methods rely on the Law of Large Numbers, which tells us that the average of the results of a large collection of sample values should approximate the expected value, and larger sample collections provide closer approximations. That is, as we increase the number of samples, we reduce the *variance* of the result, or the difference of estimated results from the true value. In rendering, variance often manifests as visual noise in the output image. A number of techniques have been developed to reduce the variance of estimates across different collections of samples.

The distribution of the samples across the domain can greatly influence the variance of the Monte Carlo estimate, which affects the rate of convergence of the estimate to the true value. Methods of generating evenly-distributed samples are discussed in Section 2.10.

A variation of Monte Carlo known as *importance sampling* was originally introduced in statistical physics (Hammersley and Morton 1954; Rosenbluth and Rosenbluth 1955), and is used today in many application areas as a method of estimating an integral. It makes

use of the fact that a Monte Carlo integral estimation process converges more quickly if instead of drawing samples from a uniform probability distribution, we draw them from a distribution shaped by an estimator function which resembles the function being integrated. The key idea is that not all values of the input variable are equally important. Some contribute more to the result than others. If we focus our attention on those values by sampling more frequently in the "important" regions, we can reduce the variance of the estimator, and the efficiency of the computational effort is increased. The samples are combined in a weighted average, with each sample's weight based on the inverse of its probability of selection. This method of weighting compensates for the distribution's nonuniformity: it is biased to favor those regions, but we compensate by weighting their output values to correct for the bias. The result is unbiased, yet focused on the "important" regions.

If a large number of sample points are used, this method produces a very accurate approximation of the integral. However, when using a large number of samples is costly or difficult, importance sampling also provides a means of achieving a good result with fewer samples. Reducing the sample count results in variance within the result – in the case of image rendering, this manifests as visual noise. Importance sampling shifts the samples toward regions of the domain where they will be most effective. Functions that have large values over small regions of the domain obtain the greatest success from this method. Importance sampling is, in short, a way to use limited resources and imperfect information to get better results than a uniform distribution by concentrating computational effort where it will do the most good.

The choice of estimator function is crucial. An estimator which provides a good approximation of the true function concentrates samples as described above and leads to more rapid convergence than a uniform distribution. Unfortunately, a poor approximation can actually make things worse. If it concentrates samples in less-relevant areas of the sample domain, variance increases, resulting in much slower convergence or none at all. The choice of estimator can even cause the Monte Carlo process to converge to an incorrect answer.

A suitable estimator must satisfy two criteria: First, it must be consistent. That is, it must converge to the correct answer, such that with an infinite number of samples the error would be zero. Second, it must be unbiased. That is, although individual errors may be non-zero, the average error must be zero. Being consistent does not imply unbiased; consider a function that converges to the correct answer from above.

2.10 Sampling

For many applications, such as the Monte Carlo methods described in the previous section, we need a source of samples scattered across the input domain. This section introduces several methods of generating these samples and discusses their characteristics.

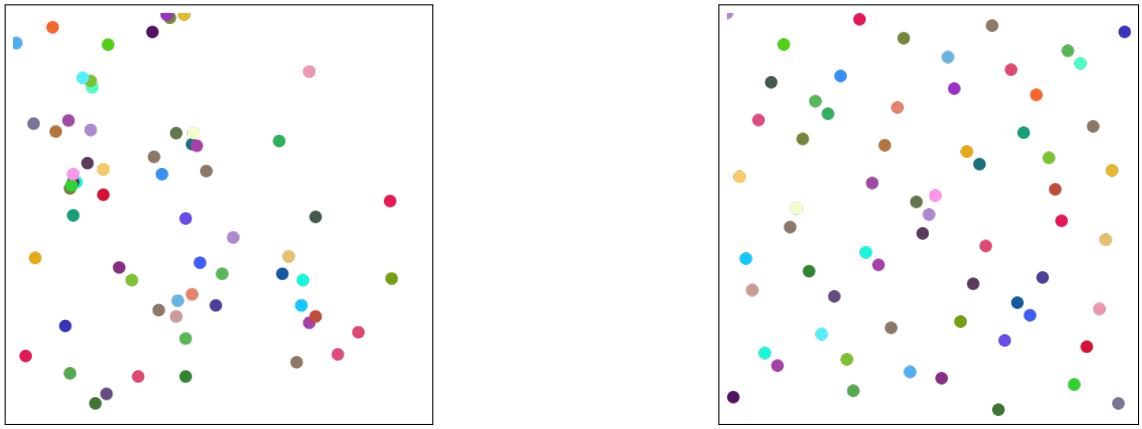
2.10.1 Random, pseudorandom, and quasirandom sequences

At first glance, randomly drawing values from the input domain seems sufficient. However, sequences of true random samples are unsuitable for two reasons: they are neither reproducible nor evenly distributed.

Reproducibility means that the sample generator can be directed to produce the same sequence of values as on a previous visit. This is not the case for true random numbers, which makes them unsuitable for applications where the same sequence of values is needed at different times or by different components of the application. For example, when running a particle simulation, we may want to generate the starting positions of the particles. If the simulation is run only once, random numbers will do, but if run repeatedly (say, for a parameter wedge), the same start positions are needed each time. Saving and reloading those positions may be impractical for large particle sets, making reproducible sample generation an important trait.

Pseudorandom sequences address this problem, providing sequences which satisfy statistical measures of randomness, and which are generated by entirely deterministic methods. Typically, pseudorandom algorithms are "seeded" with a starting value. When given the

same seed, the same sequence is produced. Additionally, pseudorandom sequences are generally quick to compute. Zafar, Olano, and Curtis (2010) compare several pseudorandom generators, including the Tiny Encryption Algorithm (TEA), which is examined in more detail in Chapter 5. However, pseudorandom sequences do not satisfy our second criterion, that the sample values must be evenly distributed.



(a) Pseudorandom (TEA) (b) Quasirandom (Sobol)

Figure 2.5: Pseudorandom and quasirandom sequences, 64 samples each, illustrating the difference in distribution quality between the two methods.

True random sequences offer no guarantees about the distribution of samples across the domain. The results can be clumpy, with clusters and gaps due to the complete lack of correlation among the samples. Pseudorandom sequences are designed to mimic the properties of random sequences, so they display the same clumpiness, as shown in Figure 2.5(a). *Low-discrepancy* sequences are designed to be evenly distributed, reducing the variance in the result by partitioning the integration domain into smaller sub-domains ("cells") that are typically lower variance than the whole. Figure 2.5(b) illustrates the result. These methods are used to improve integration convergence within the samples that will be averaged for a single cell (Kollig and Keller 2002). Recent work has pointed out that the lack of correlation between cells in Monte Carlo rendering also affects the perception of noise in the image, and demonstrated use of a global distribution of samples that are low-discrepancy between pixels as well as within each pixel (Georgiev and Fajardo 2016; Kensler 2016).

2.10.2 Low-discrepancy sequences

Discrepancy is a measure of the uniformity of a set of points in N dimensions, effectively comparing the Monte-Carlo estimate of the volume of an axis-aligned region with the actual volume. Shirley (1991) and Dobkin et. al (1996) both argue for using discrepancy to evaluate sample distributions for rendering, even those that are not normally described in terms of their discrepancy.

Given a set of points in the s-dimensional unit hypercube, $P = p_1, ..., p_N$, the *star discrepancy* is defined as

$$D_N(P) = \sup_{B \in F} \left| \frac{C(B)}{N} - V(B) \right|, \tag{2.5}$$

where V is the volume of box B (its Lebesgue measure), $C(B)$ is the count of points in P that fall in B, and F is the set of s-dimensional boxes with one corner at the origin:

$$\prod_{d=1}^{s} [0, b_d) \; : \; 0 \leq b_d \leq 1. \tag{2.6}$$

A *low-discrepancy* set is evenly spread over the sampling interval, producing no large gaps or tight clusters. These sets are widely used as an alternative to sequences of uniformly-distributed random numbers, because they share some desirable properties of random variables while offering more reliably even coverage of the sample space, thus algorithms using such sequences may have better convergence.

Several methods of constructing low-discrepancy point sets have been developed. Although not normally described in terms of their discrepancy, *Poisson Disk* and *Blue Noise* provide low-discrepancy point sets (Shirley 1991; Dobkin, Eppstein, and Mitchell 1996). Methods to construct these sets build or optimize the entire set in a preprocess. One class of methods are dart throwing algorithms (Dippé and Wold 1985; Cook 1986; Lagae and Dutré 2008) which place points iteratively, enforcing a minimum N-dimensional distance between points by discarding candidates that are too close to previous points. Another class of methods are iterative relaxation algorithms, which randomly initialize the entire

point set, then displace points according to constraints. These include Lloyd's method or capacity-constrained Voronoi tessellation (Lloyd 1982; Balzer, Schlömer, and Deussen 2009). Georgiev and Fajardo (2016) propose using a blue noise sequence to correlate nearby samples for a path tracing renderer, and improve image quality, especially for low sampling counts. Similar results can be achieved using a progressive Sobol sequence, providing better scaling with sample count and number of dimensions.

Jittered or *Stratified* sampling divides the domain into N cells and places a point or points at random position(s) within each cell. *Uncorrelated Jitter* (Cook, Porter, and Carpenter 1984) is straight stratified sampling in each dimension or partition of dimensions. As a low-discrepancy method, with one point per cell it can have clumps or gaps at cell boundaries. Extended to multiple points per cell, the points within a cell will have the same properties as uncorrelated random numbers. *Latin Hypercube / N- rooks* (Shirley 1991) jitters samples in each dimension independently, then shuffles each 1D set and combines dimensions in random order. Results are evenly distributed in each dimension but can clump and gap in N-dimensional space. *Multi- jittered* sampling (Chiu, Shirley, and Wang 1994) combines N-rooks and jittered-stratification, and *Correlated Multi-jittered* (CMJ) sampling (Kensler 2016) addresses correlation between pixel samples, but primarily for two sampling dimensions. This author of the CMJ paper discusses the advantages of Sobol sequences over the CMJ approach since Sobol does not fix the number of points a priori and gives a progressive increase in density as new points are added from the sequence. The Cell-Indexed Sobol method described in Section 5.3 provides the benefits of correlation between pixels with these advantages of Sobol sequences.

Finally, we have the *Quasirandom* methods, deterministic algorithms for generating sequences satisfying the properties of low-discrepancy points. The name is derived from the similarity to pseudorandom number algorithms, which are deterministic algorithms for generating sequences satisfying the properties of random numbers. Unlike most pseudorandom number algorithms, most quasirandom algorithms (including Sobol) can be accessed

in arbitrary order, or by multiple parallel computations, since they can generate the Nth number in the sequence without generating all the preceding numbers.

The *Radical Inverse* quasirandom methods operate on numeric representations of the sequence index. The Van der Corput sequence (Hammersley 1960) is constructed by reversing the digits of the base-n representation of the sequence index and dividing the result, called the radical inverse, by the corresponding power of n. This sequence partitions the unit interval into successively finer segments, placing sample points within each segment in the same pattern. High-dimensional Halton (1964) sequences (in practice, above 6-8) are unsatisfactory, as their large prime-number bases lead to correlation between dimensions, but this flaw can be ameliorated by scrambling or shuffling methods (Owen 1995; H. Faure 1992)

The Faure (1982) and Sobol (1967) sequences are similar to the Halton sequence in that they partition the unit interval into successively finer segments and populate them uniformly, but they use only one base, then reorder the coordinates in each dimension. Faure uses as its base the smallest prime number which is less than or equal to the dimensionality of the problem, but is significantly slower than Halton and Sobol (Galanti and Jung 1997).

The Sobol sequence uses only base 2, but uses a more complex reordering method based on a set of special binary values called *direction numbers*, from which the sequence values are generated. Poor selection of direction numbers can produce inefficient Sobol sequences; however, Joe and Kuo (2008) provided sets of direction numbers which produce Sobol sequences satisfying additional uniformity conditions. Antonov and Saleev (1979) developed a variant which uses Gray codes to reduce the computation further, so can update $x_{n-1} \rightarrow x_n$ with only a single XOR in each dimension.

Chapter 3

PROJECT OVERVIEW AND CONTEXT

ComputeGlory is an OpenGL application for rendering of wavelength-dependent atmospheric phenomena using GPU-accelerated Mie scattering calculations. The overall application is not intended as a demonstration of the best possible performance from a purpose-written Mie scattering renderer; rather it is a research testbed, a framework for comparing multiple variations of the components which collectively generate and render an image of a glory. The framework is modular, with several phases which can select among multiple implementations. The categories of techniques explored here include scattering algorithms, approximations to Mie scattering, wavelength and scattering angle selection, pseudo- and quasirandom sample sequences, image generation methods, incremental rendering, and adaptive step-sizing.

I demonstrate the accuracy, speed, and utility of the various implementation choices and techniques using glories as the primary example. I also discuss applicability to other phenomena which can be rendered using the same overall application framework. The most significant differences among these phenomena lie in the number of Mie scattering calculations required and how the inputs of those calculations are determined; and in the assumptions and mathematical form of the light scattering calculations.

The application is not merely an implementation of Mie scattering. It includes a pipeline of OpenGL compute shaders which convert the wavelength-oriented results of Mie scattering calculations (wavelengths and corresponding intensities) into displayable RGB colors, accumulate and blend the results into pixel values, and generate the final displayable texture. Finally, standard OpenGL vertex and fragment shaders perform final rendering of the glory to screen. Optionally, the glory is rendered with additional environmental features

to further support the realistic appearance of the glory as seen by human observers, such as simulated cloud haze.

3.1 Development environment and hardware specs

All development and data collection was performed on a 2014 Dell Alienware laptop running Windows 7 Home Premium (64-bit), using Visual Studio 2012.

The development machine's CPU was Intel Core i5-4200M CPU @ 2.50GHz with 2 processor cores and 8GB RAM. Its GPU was an NVIDIA GeForce GT 750M (Kepler architecture) which supports OpenGL versions through 4.5.

3.2 System structure

The ComputeGlory application comprises several components: a C++ host program, which provides the user interface and controls the GPU activity; a suite of compute shaders running on the GPU; standard OpenGL rendering shaders for final image display; and auxiliary data in the form of external data files, hardcoded data tables, and a reference solution which is generated on the fly.

3.2.1 Host program

The C++ host program, called ComputeGlory, provides a GUI which allows the user to set parameters and select among the multiple organizational and algorithmic variations. Using ComputeGlory, the user can:

- run complete glory renderings
- view images, timing, and error as rendering progresses
- save intermediate and final images along with detailed logs and data files

- compare rendered results visually and numerically to previous results or to ground truth using a variety of error metrics

The foundation of ComputeGlory comes from the OpenGL Graphics and Compute Samples Pack provided by NVIDIA's GameWorks library, which can be found online at http://docs.nvidia.com/gameworks/index.html. This is a collection of sample graphics applications which demonstrate individual advanced features of OpenGL. The samples pack includes the source code for a simple application framework used by all the samples, which provides GL window and context management, a main event loop, user input handling, rendering, initialization, and shutdown, along with specialized asset loaders for shaders, textures, and models. The NvSampleApp class adds a simple GUI library, NvTweakBar, to this framework. The sample applications are all subclasses of NvSampleApp, and ComputeGlory is derived from this class as well. I implemented several extensions to NvTweakBar to support large collections of parameters with complex interdependencies. I also extended the GLSL loader to add #include functionality akin to that found in C/C++ to support code reuse and reduce redundancy among the compute shaders used by ComputeGlory. Although OpenGL has an "include" mechanism, it is not the familiar verbatim splicing of text from an external file, as GLSL is not directly aware of the filesystem in its surrounding environment.

The NVIDIA samples pack relies on two external, third-party open source libraries: GLEW (http://glew.sourceforge.net/), a C/C++ library which provides efficient and simplified access to OpenGL extensions; and GLFW (http://glfw.org), a C library which supports window creation & management and user input.

3.2.2 OpenGL compute shaders

High-level rendering flow

Once the user has set the physical, algorithmic, and execution-control values needed by ComputeGlory the main computations are carried out by a collection of OpenGL compute

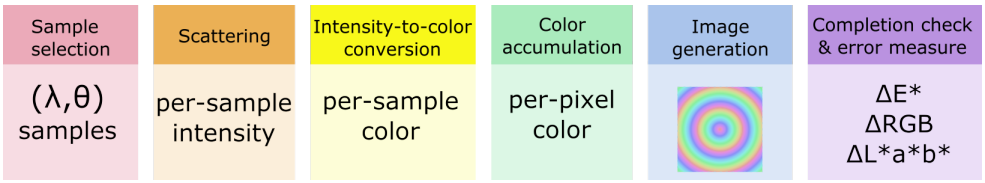


Figure 3.1: Phases of compute shader pipeline and information produced in each phase.

shaders (see Sections 2.7 and 2.8 for background) which work together to perform all the calculations necessary to create the glory image. Rendering the final image to the screen is performed by OpenGL graphics pipeline vertex and fragment shaders.

Figure 3.1 gives an overview of the compute-shader workflow, showing the phases of the workflow in the order that computation proceeds through the phases. The lower portion of each box indicates the information produced by that phase. Table 3.1 addresses the code structure of the compute shader phases. The left column lists the phases and the right column lists the individual compute shaders used in each phase. Some supporting functions are used by multiple shaders, and have been extracted into separate files for inclusion by those shaders. For example, in the image generation phase, `sweepFull.glsl` and `sweepSplit.glsl` are two variants of the same shader, with their common functionality in `sweep.glsl`. In each phase, I have separated variants into two (or more) shaders so that selecting a variant is done once, by the host program, rather than within each invocation of a combined shader.

Chapters 4 through 7 explore the various options and decisions made for each of these phases. Each chapter includes discussion of the tests that were run to compare these options, and test results indicating the most effective choices.

Sample Selection (wavelength/angle pairs)

The Mie scattering function takes a pair of values as input: the wavelength λ of the incident light and the scattering angle θ to consider. To render atmospheric phenomena, this function must be called many times, with many (λ, θ) pairs. In the ComputeGlory applica-

Phase	Compute shaders and supporting GLSL files
Sample selection	wavelength, wavelengthStruct sobol_data, TEA
Scattering	scatter-up, scatter-down, scatter-up2D, scatter-approx bhmie, complex
Color conversion	convertColor, convertColor-splat colorFunctions solar_irradiance
Color accumulation	accumulateColor, mergeColor
Image generation	sweep, sweepFull, sweepSplit blend, bufToTxt
Completion check & error assessment	convertImageRGBtoLAB, errorTxt

Table 3.1: Compute shaders and files

tion, selecting these pairs is not a distinct phase. Instead, this functionality is separated into GLSL functions which the scatter, color-convert, and accumulate/merge shader phases call to determine the appropriate scattering angle and wavelength for each shader invocation based on the index assigned to that invocation by the OpenGL system.

Chapter 5 discusses the methods used to generate random values used to select the (λ, θ) pairs. This chapter also discusses my investigation into various implementation options for the Sobol quasirandom sequence.

Chapter 6 discusses the overall organizational scheme: 1D, in which the scattering angles are determined by the pixel location being rendered, and a group of wavelengths is generated for each angle; or 2D, in which a group of (λ, θ) pairs is sampled from the 2D (wavelength x scattering angle) domain.

Chapter 7 looks at the manner in which the (λ, θ) pairs are grouped during the course of the render: all in one large group, or in smaller cohorts with intermediate results. This chapter also explores the tradeoff between speed and accuracy which results from incrementally producing the glory image, beginning with an approximation that can be rendered in one animation frame's time budget and performing the full Mie scattering calculations over several subsequent animation frames.

The size and organization of the intermediate buffers for each phase depend on all these aspects of the rendering process, as do the parallel structure and group size of the shader dispatches. These issues are discussed in the corresponding chapters.

Scattering

The scattering phase uses the Mie scattering equations to calculate the scattered light intensity for each (λ, θ) pair in the current cohort.

Section 4.1 demonstrates the speedup achieved by executing Mie scattering in a parallel GPU format using compute shaders, as compared to a serial CPU implementation of the same calculations.

Section 4.3 introduces the different forms of the Mie algorithm, their relative efficiencies, and their applicability to rendering atmospheric glories.

Color Conversion

This phase converts the Mie scattering results from (λ,intensity) pairs to the corresponding CIE XYZ tristimulus values using the multi-lobe fit functions from Wyman, Sloan, and Shirley (2013). At the same time, it scales the intensity values by the measured intensities of the corresponding wavelengths in the solar spectrum, as the atmospheric phenomena are caused by incident sunlight, which is not evenly distributed. Finally, it converts these results to displayable RGB values using the methods and matrix values from (Lindbloom 2017a).

In 1D pixel-oriented mode, each invocation of this phase processes all the wavelengths for a given pixel, so it accumulates the result for each wavelength. Then it converts the aggregate XYZ value to display RGB. This kernel is called once per pixel of the high-resolution slice and writes its output to another OpenGL shaderbuffer.

In 2D-sampling mode, the color conversion shader receives a buffer containing scattering angle for each item, as well as wavelength and intensity. After converting to RGB,

the resulting value is splatted into the shared output buffer at the sub-pixel position corresponding to the scattering angle.

The 1D and 2D variants of this phase are implemented as separate compute shaders. They share common code found in a separate GLSL file, `colorFunctions.glsl`.

Accumulate / Merge

During an incremental render (see Chapter 7), this phase merges the color values produced by the color conversion shader into the previous incremental results. The blend weight for the new values is based on the number of Mie calculations in the just-completed cohort as a proportion of the total number of Mie calculations performed so far.

Image Generation

When ComputeGlory is using a high-resolution radial slice (see Section 4.2), it builds a full-circle image by sweeping the slice in a circle and using Gaussian importance sampling to blend multiple slice pixel values for each pixel of the output image.

The next two shaders together perform a 2D symmetrical Gaussian blur at each pixel of the final image. Because the convolution kernel is symmetrical, it can be separated into two 1D kernels executing vertical and horizontal passes. This reduces the $O(n^2)$ 2D convolution to two $O(n)$ operations for greater efficiency with no loss of accuracy.

The final compute shader of this phase performs data conversion from the shaderbuffer format used for general data interchange by the compute shaders into an OpenGL RGBA texture to be used by the standard rendering pipeline.

Completion Check / Error Assessment

At several points in the rendering process, the current image is compared against a reference solution, a previously-rendered image which was generated using the highest image resolution and slice resolution possible on the development machine. This image is generated

with 800 scattering wavelengths per pixel, well above the theoretical required minimum number for a stable result. This is used as the "ground truth" image for error calculations. The reference image is discussed in more detail in Section 3.2.4.

The error shader compares the most recently rendered image against the reference solution according to multiple error values and produces difference images in each metric. (See Section 3.3.2 for discussion of the error metrics.) A second shader processes these images to report average error and sum-of-square-differences.

During an incremental render (see Chapter 7), error calculations are performed at the end of processing each group of (λ, θ) pairs. If error is specified as the end criterion for the render, the intermediate error results are examined to see whether the render should terminate or continue. Error is also calculated and reported at the end of the render, whether incremental or single-frame. Finally, when a render has completed, the user can request to perform comparison against the reference solution image and select one of the error images to display for visual assessment.

3.2.3 OpenGL rendering pipeline shaders

When an image is ready to display, ComputeGlory resumes treating the GPU as a graphics device and uses OpenGL's standard rendering pipeline to display results to the screen. Simple pass-through vertex and fragment shaders receive the texture produced by the image-generation compute shaders and apply it to a screen-aligned quadrilateral with no further modification. The image data is already on the GPU as an OpenGL texture and never needs to be transferred back to the CPU, so the switch from behind-the-scenes physics calculation and image processing to onscreen rendering incurs no further overhead.

3.2.4 Reference solution

Error calculations are performed at several points in the rendering process and on demand at the end of a render. The current image is compared against a reference image, which

is the output of a previously-generated high-resolution, high-sample-count rendering. The reference image was rendered using a set of image and physical parameters which are considered the "standard" parameter set and are used for all other tests in this dissertation unless indicated otherwise. This is a 1024x1024 image using the WvPx sample selection method at 800 equally-spaced wavelengths per pixel, well above the theoretical minimum requirement. Physical parameters are: droplet radius 10 microns, index of refraction 1.33 (water with no absorption), field of view 4.5° from the glory axis, which runs through the center of the image.

The reference solution image is shown in Figure 3.2. It does not include the central white disc which is the visual result of the sun being a disc rather than a point light source, as this work is primarily concerned with the color banding surrounding the central disc. Additionally, in actual glory viewings, the center of the image is often partially filled by the Brocken spectre, which is the shadow of the observer and any surrounding vehicle or structure. The reference image also consists of the glory alone; it is not composited with a background of a cloud or fogbank, as the specifics of those images and the compositing process would depend on the particular application in which the glory rendering is to be used.

3.2.5 Auxiliary data

ComputeGlory makes use of several external data sets. Some are loaded by the host application at run time from auxiliary files and passed to the compute shaders. Datasets handled in this manner includes solar spectrum intensity data (Harder et al. 2005), 22KB; precomputed Sobol lookup tables, ranging from 4KB to 22KB depending on lookup index size (see Section 5.5); and CDF lookup data (output from a previous high-resolution rendering), 1138 KB.

Others are included into the compute shaders when the shaders are loaded and compiled. The direction numbers for standard Sobol sequence generation (Joe and Kuo 2008)

Figure 3.2: 1024x1024 reference solution computed with 800 wavelengths per pixel (WvPx sample selection method). (Northeast quadrant shown.)

and Cell-Indexed Sobol (Olano and Blenkhorn, in preparation) are incorporated by the OpenGL shader compiler in this way (8 KB total).

3.3 Assessment / Measurement

This section presents the variety of assessment and measurement techniques employed during this work. These techniques are used for evaluation of the costs and benefits of proposed glory rendering methods. The error metrics described in Section 3.3.3 are also used for controlling the rendering process directly when computed as part of an incremental render (Chapter 7).

3.3.1 Internal and external assessment

To assess the visual quality of the results of this system, both external and internal comparisons are necessary. External comparisons use existing best solutions as ground truth. Although there are other implementations of Mie scattering which render images of glories, they are available as executables only. Therefore, it's not possible to isolate the portion

of these applications which is equivalent to particular phases of the ComputeGlory shader pipeline. Only coarse end-to-end wall-clock execution timing can be performed. Instead, I have chosen to use my own implementation of the widely-used BHMIE algorithm, which forms the basis for both MiePlot and IRIS. My implementation adheres as closely as possible to the original (Bohren and Huffman 1983).

Internal comparisons demonstrate the relative quality of the results as we vary parameters and other aspects of the rendering process. These include:

- Speed comparison of algorithmic variants of Mie scattering
- Comparison of the total number of Mie scattering calculations required to produce equivalent quality images for different sample selection strategies.
- Image quality and speed comparison between renderings with different control parameters and organizational structures
- Speed comparison of implementation variants of Sobol sequence generation.

3.3.2 Visual and numerical image quality

To compare rendered scattering images, I used a perceptually uniform color distance metric. *Perceptually uniform* means that if two pairs of colors are determined to differ by the same value of the metric, then a human observer will perceive the pairs of colors as being equally far apart. The metric I have used throughout this work is the 1994 formula of ΔE^* as defined by the International Commission on Illumination (CIE) (Fairchild 2005). The 1994 formula addresses some perceptual non-uniformities that were considered objectionable in the original 1976 formula. ΔE^* works by, in effect, transforming the colors into a perceptually-uniform color space and calculating their Euclidean distance. Details of ΔE^* and the L*a*b* color space are given in Section 2.4.

Using ΔE^* to calculate per-pixel errors, the mean square error between two images provides an objective and perceptually-grounded image-to-image comparison value.

Listing 3.1: Using the error shader and utility functions.

```
runErrorTxtShader(fullMieTexture, finalTexture,
                  errorTextureDeltaE, errorTextureRGB,
                  errorTextureLAB);
float avgError = averageErrorFromTxt(errorTextureDeltaE);
float rmsError = rmsErrorFromTxt(errorTextureDeltaE);
```

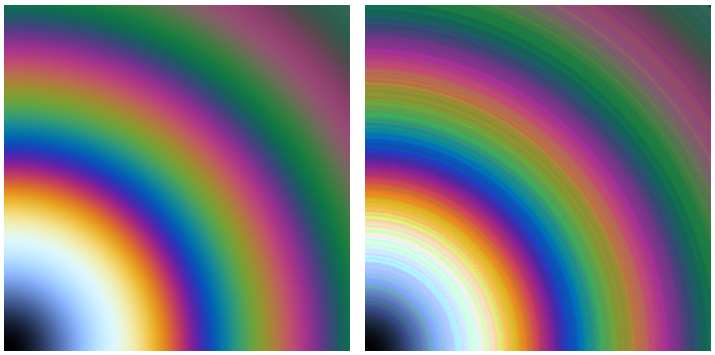
3.3.3 Error: ΔE^* error and comparison image

In Listing 3.1, fullMieTexture is the reference solution image described in Section 3.2.2. The errorTxt compute shader compares a candidate texture against the reference solution using three different error metrics: ΔE^*, per-pixel RGB difference, and per-pixel difference after both images are converted to L*a*b*. (See Lindbloom (2015, 2017a, 2017b) for more details.) Figure 3.3 shows the reference solution, the final image from a rendering with a deliberately low wavelength count in order to produce visible errors, and the three error images. The error images have been brightened and contrast-enhanced for visibility. Brighter regions indicate higher values, i.e. greater error. The Mean and RMS errors for each metric are shown in Table 3.2.

The L*a*b* error image is particularly interesting. Overall, it is primarily blue and green, indicating that errors are primarily in the color being displayed, rather than in the luminance, which is rendered in the red channel. However, red dominates in the lower-left corner of the image, which corresponds to the center of the glory. Looking at the reference solution and the final rendered image, we see that this is the black-to-gray region of the glory. The final image varies from the reference solution only in brightness.

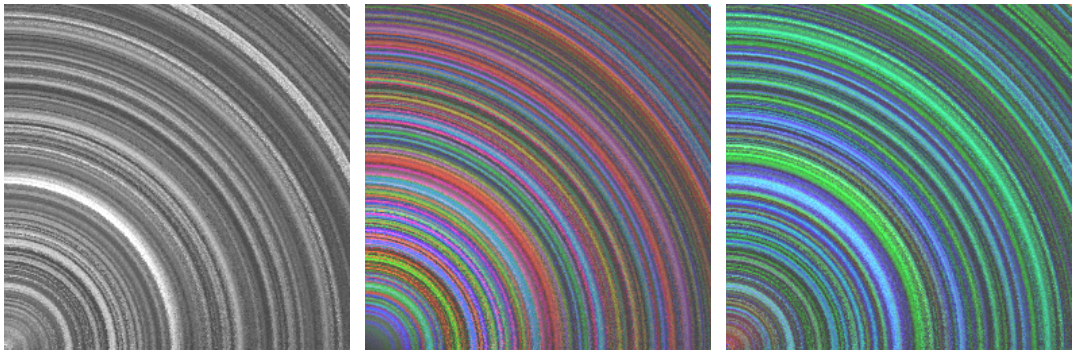
ΔE^* average	ΔE^* RMS	RGB average	RGB RMS	L*a*b* average	L*a*b* RMS
0.0299	0.0108	0.0262	0.0101	0.0080	0.0056

Table 3.2: Image comparison values for example images of Figure 3.3



(a) reference solution (comparison)

(b) final image



(c) ΔE^* error

(d) RGB difference

(e) L*a*b* difference

Figure 3.3: Reference solution comparison image, final render, and enhanced error / difference images. (Northeast quadrants)

3.3.4 Timing

To assess the speed of this system, it is not enough to ask whether it is real-time, as this can have many interpretations. The minimum goal is to complete a full cycle – scattering, color-transformation, image generation, and rendering – in no more than 33 ms. This equates to 30 frames per second (the usual description of real-time), but only if the system is doing nothing else.

The primary method of measuring computation time for the ComputeGlory application is GPU shader timing. Each phase of the compute shader pipeline is timed separately and reported after completion.

In addition to measuring the precise amount of computation time consumed by the GPU, it is also useful to assess the total elapsed time a user must wait from pressing “go”

to seeing the final rendered image. The ComputeGlory application is a development framework structured to offer flexibility and choices for comparisons between techniques, and as such the wall-clock timing results produced by this application include time spent on activities which would be irrelevant or undesirable in a production setting. These results should not be interpreted as directly representative of the overall performance that might be expected if the same core shaders were integrated into a tailored and streamlined application that uses only the most efficient techniques and does not concern itself with intermediate user feedback or post-render analysis. Still, it is informative to see that one ComputeGlory render completes in a few seconds and another, with identical results but different algorithmic choices, takes twice as long.

As the intended eventual use of this work is to produce a stream of perceptually-accurate images for a human viewer in an interactive computer graphics application, rather than physically-accurate single images for atmospheric physics applications, an additional definition of "rendering speed" arises. In Chapter 7, I discuss a method for incrementally producing a glory image over the course of multiple animation frames, beginning with an approximation that can be rendered in one frame and spreading out the thousands of Mie scattering calculations over the following animation frames. Given this structure, we can also ask how many frames of animation are required to complete the render with an output image of a desired level of accuracy, while staying within our time budget?

Additional questions explored in the indicated chapters:

- Where are the bottlenecks, i.e. which phases of the shader pipeline consume the most time? (Scatter, then color conversion – see e.g. Section 4.2)
- How much time does the Mie scattering phase consume, and how does this vary with the physical parameters (see Section 4.3)?
- How much faster is my initial GPU reimplementation of BHMIE compared to a CPU implementation (see Section 4.1)?

- How much speedup is achieved by using a radial slice compared to rendering each pixel of the output image (see Section 4.2)?
- How do 2-dimensionally sampled Mie calculations perform compared to pixel-oriented calculations where only the wavelengths are sampled (see Chapter 6)?

Finally, we need to consider not only whether this system is better than previous efforts, or how much better, but *why* is it better. We need to understand where the benefit is coming from. Specifically, how much of the benefit comes simply from running on the GPU, and how much comes from the parallel restructuring of the scattering code, and how much comes from algorithmic enhancements such as 2D Sobol sampling. Chapter 8 compares results from existing CPU-based methods, several intermediate versions of my GPU implementation, and my final version. Overall findings and recommendations are discussed.

3.3.5 Timing: GPU (shader time) for compute shader phases

Elapsed-time measurements for the OpenGL compute shaders are performed by OpenGL timer queries, which record time intervals with nanosecond precision. These queries have been part of core OpenGL since version 3.3 in 2010. As illustrated in Listings 3.2 and 3.3, timer queries can be used in two ways.

Listing 3.2 shows the `GL_TIMESTAMP` form, which is used to record a specific time. By pairing two timestamps at the start and end of a block of GPU operations, the elapsed interval is obtained. The calls to glQueryCounter (lines 2 and 11) request that the timestamps be written into the specified queries. As this is an asynchronous action, the while loops (lines 3 and 12) check the state of their respective queries until the timestamps have been recorded. Finally, lines 17-19 retrieve the timestamp values and calculate the interval.

Listing 3.3 shows the `GL_TIME_ELAPSED` form. This is used to time a scoped block of GPU operations. It reports actual execution time, beginning when the first operation in the scoped block starts executing and ending when the last one completes. Line 1 establishes

Listing 3.2: GPU timing with GL_TIMESTAMP

```
available = 0;
glQueryCounter(queries[0], GL_TIMESTAMP);
while (!available) {
  glGetQueryObjectiv(queries[0], GL_QUERY_RESULT_AVAILABLE,
              &available);
}

   [other operations]

available = 0;
glQueryCounter(queries[1], GL_TIMESTAMP);
while (!available) {
  glGetQueryObjectiv(queries[1], GL_QUERY_RESULT_AVAILABLE,
              &available);
}

glGetQueryObjectui64v(queries[0], GL_QUERY_RESULT, &timeStart);
glGetQueryObjectui64v(queries[1], GL_QUERY_RESULT, &timeEnd);
elapsedTime = (timeEnd - timeStart)  / 1000000.0;
```

Listing 3.3: GPU timing with GL_TIME_ELAPSED

```
glBeginQuery(GL_TIME_ELAPSED, queries[x]);

   [other operations]

glEndQuery(GL_TIME_ELAPSED);
glGetQueryObjectui64v(queries[x],
                  GL_QUERY_RESULT, &timingResults[x]);
```

the start of the scope and identifies the query in which the timing result should be stored. Line 5 ends the scope, causing the OpenGL system to record the elapsed interval and mark the query as available. Lines 6-7 retrieve the interval duration and store it in a local array.

Because the `GL_TIME_ELAPSED` timer queries are scoped, only one can be active at a time. When multiple nested or interleaved timing operations are needed simultaneously, the `GL_TIMESTAMP` form provides the desired flexibility although the host code is responsible for more bookkeeping and calculations.

3.3.6 Timing: end-to-end wall clock time for entire run

Elapsed-time measurements in the C++ host program are performed by the `steady_clock` timer class, which records time intervals with nanosecond precision. `steady_clock` is one

Listing 3.4: Using the `steady_clock` class to time small intervals.

```
1  wallClockStart = steady_clock::now();
2  wallClockEnd = steady_clock::now();
3  duration<double,std::milli> time_span =
4      wallClockEnd - wallClockStart;
```

of three clock types offered by the `std::chrono` library, which was introduced in C++11. The other two are `system_clock` and `high_resolution_clock`.

The choice of `steady_clock` for this project was determined by the selection of Visual Studio 2012 as my development environment. In all versions of Visual Studio through 2013, `high_resolution_clock` was an alias for `system_clock`, which is based on the real-time clock of the system and does not guarantee the resolution or steadiness (monotonicity) needed for timing small intervals. Beginning with Visual Studio 2015, `high_resolution_clock` is an alias for `steady_clock`, which has itself been updated to meet C++ standards requirements for steadiness and monotonicity (Microsoft 2018).

Listing 3.4 shows typical usage for this class. The template specification on line 3 indicates the desired units (`std::milli`) for the reported value, so that it's not necessary to convert units everywhere this class is used. The duration value is recorded internally at nanosecond precision.

Chapter 4

SUPPORTING CONTRIBUTIONS

4.1 Serial CPU implementation vs. parallel GPU implementation

The Mie scattering equations yield the scattering intensity for a single wavelength and scattering angle. The calculations for one such input pair are entirely independent of the calculations for any other input pair. When more than one Mie scattering calculation is to be performed, they can be organized and executed in any order. Furthermore, they can be executed simultaneously with no dependencies or interference between calculations. The independence of the calculations affords the opportunity to perform groups of Mie calculations in parallel. OpenGL compute shaders are well-suited to this structure.

Early Mie algorithms such as DBMIE (Dave 1968) and BHMIE (Bohren and Huffman 1983) executed as serial programs, with the main loop examining a range of scattering angles for a single wavelength. Later versions such as Wiscombe's (1979) MIEV0 and MIEV1 took advantage of special vector hardware such as presented by the Cray-1 supercomputers to transform the loop into simultaneous processing of a vector of scattering angles. However, they retained the one-wavelength-at-a-time structure of BHMIE.

In this chapter, I show that computing scattering results in parallel on the GPU, using numerous invocations of an OpenGL compute shader which performs Mie scattering for a single (λ, θ) pair, is more efficient than the serial CPU implementation of the same Mie calculations. Both versions are re-implementations of the main BHMIE calculations, drawing from MIEV0 and Flatau's C translation, `bhmie-c` (Flautau 1998). Both are publicly available from http://scatterlib.wikidot.com/codes under the Creative Commons BY-SA 3.0 License[1]. To compare the effects of the two implementations of the BHMIE algorithm, I

[1] https://creativecommons.org/licenses/by-sa/3.0/

replicated the structure of bhmie-c as closely as possible. In particular, I did not update the code to take advantage of modern multi-core CPUs. Therefore, these results should not be interpreted as representing the best possible CPU performance for these calculations.

Both the CPU and GPU implementations use upward recurrence to compute A_n, the logarithmic derivative of ψ_n (discussed in Section 4.3). They are called with identical physical parameters, such as the refractive index for water, and they write to the same data buffer for use by subsequent stages of the rendering compute-shader pipeline.

The elapsed-time results given in the next section are for the innermost section of each implementation, in which individual Mie calculations or groups of shader invocations are executed and results are written to the data buffer. These times do not include overhead time such as passing buffers and control values to the GPU. I recorded these times as well and found the differences from the inner-loop timing results were trivial. As the setup time is dependent on the environment and structure of the host application, I have chosen to focus on the timing results for the core calculations.

Configuration details for the CPU and GPU used in this test were given in Section 3.1.

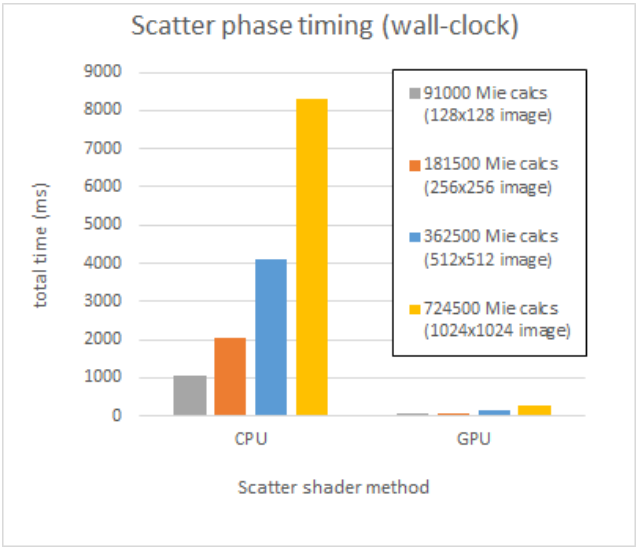
4.1.1 Results: Timing

Table 4.1 and Figure 4.1 show timing comparisons for Mie scattering as discussed in the previous section. Timing data was collected for image sizes from 128x128 to 1024x1024, using the radial slice method described in Section 4.2 at 2x resolution, which gives equivalent Mie scattering calculation counts as indicated in the charts. These timing measurements were generated using 1D pixel-based wavelength selection at 500 wavelengths per pixel, with wavelengths evenly spaced across the visible spectrum. All times are reported in milliseconds. These are averages over five runs.

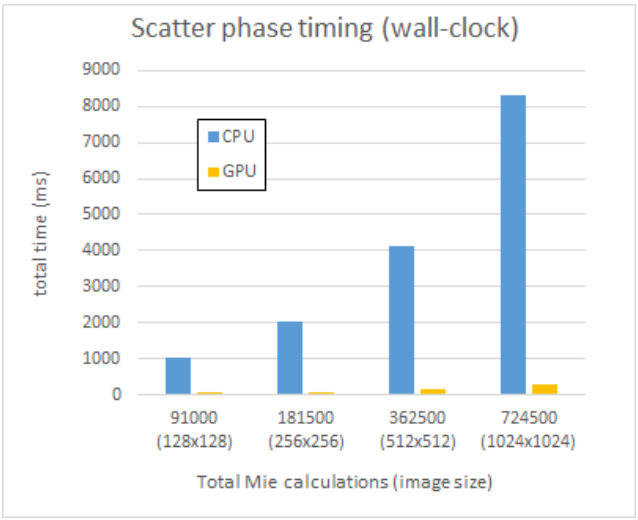
For all image sizes, GPU scattering is much faster. Figure 4.2 shows the speedup between CPU and GPU. Speedup is not as pronounced for the smallest image size as for the others, presumably due to the overhead of moving data to the GPU and invoking the com-

	128x128	256x256	512x512	1024x1024
CPU	1048.86	2044.92	4104.43	8308.48
GPU	60.40	77.00	142.81	282.62

Table 4.1: Timing results (milliseconds) for scatter phase using upward-recurrence Mie scattering on CPU vs. GPU compute shader.



(a) CPU / GPU



(b) image size

Figure 4.1: Timing results for CPU vs GPU scatter phases

pute shader. Even with the largest image sizes and highest Mie calculation counts applied in this work, the GPU is not being fully utilized, but even partial utilization demonstrates a clear benefit.

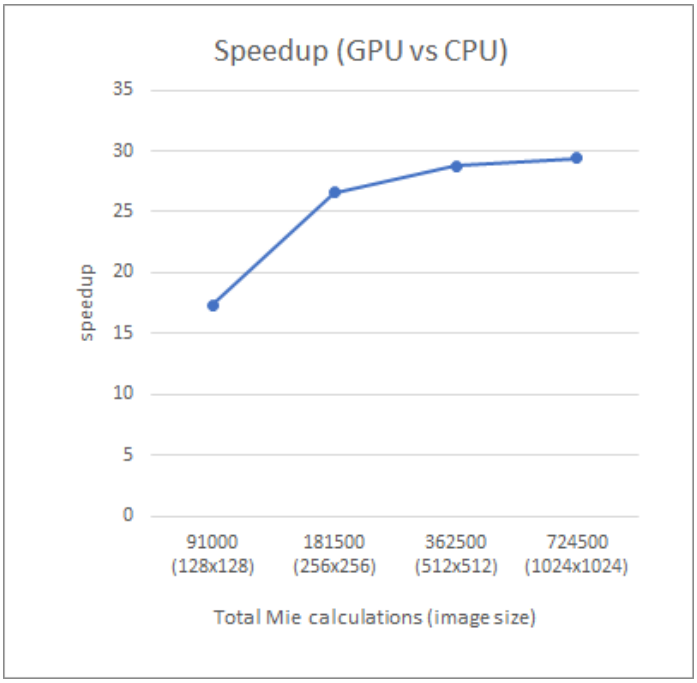


Figure 4.2: CPU vs. GPU speedup

4.2 Radial slice optimization

The glory is a perfectly symmetrical circular phenomenon, given the assumption that the cloud droplets are uniform in size and composition. The scattering behavior of a given wavelength of light calculated for one point of the glory also applies to all other points at the same apparent radius from the center of the glory, since all such points lie at the same scattering angle from the center. The same is true for the other atmospheric phenomena addressed by this work.

4.2.1 Leveraging radial symmetry

The circular symmetry affords the opportunity to reduce redundant calculation by calculating scattering intensities along a single radial slice and sharing those results among pixels which have the same scattering angle (the same radius from image center). However, because the image is a discrete grid, most pixels do not align with the pixels of this slice, so their values must be determined as a blend of a small neighborhood around the corresponding slice location. In this chapter, I show that computing scattering results for a radial slice of the glory, then resampling into a final circular image, is more efficient than methods based on pixel-to-pixel exact copying. I illustrate that a slice at the same resolution as the final image is too coarse and produces images with unacceptable errors, and I show that increasing the resolution to 3x is sufficient. Finally, I discuss the Gaussian reconstruction kernel and blending method selected to resample a radial slice into a full-circular image.

4.2.2 Slice resolution

The final render to screen uses a full-size, single-resolution RGBA texture.

As previously discussed, we do not actually need to perform Mie scattering calculations for each pixel in the output image. There is a great deal of redundancy due to the radial symmetry of the glory phenomenon. A simple optimization is to perform full calculations at the pixels of one half of the image and copy the resulting color values to the other half of the image. With a little more bookkeeping, we can compute only one quadrant, and copy to the other three quadrants; or eighths (45˚ wedges); and so on. However, we can take this progression only so far if we wish to compute, for every output pixel, a source pixel that falls at exactly the same scattering angle.

Instead of seeking the minimum region of the image that gives us exact correspondences to pixels in the rest of the image, the new goal is to compute several nearby pixels and blend them. We perform full Mie calculations for a radial slice, the pixels lying along one half-diagonal of the image as shown in Figure 4.3, then sweep this radial slice around a

full circle to generate the complete glory image. However, the actual pixel positions along the half-diagonal correspond to scattering angles more widely spaced than the pixels of the original image by a factor of $\sqrt{2}$, as can be seen by comparing the red and black dots in Figure 4.4.

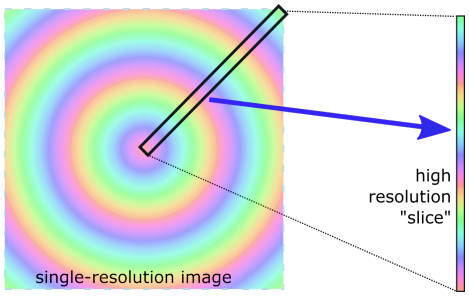


Figure 4.3: Radial slice

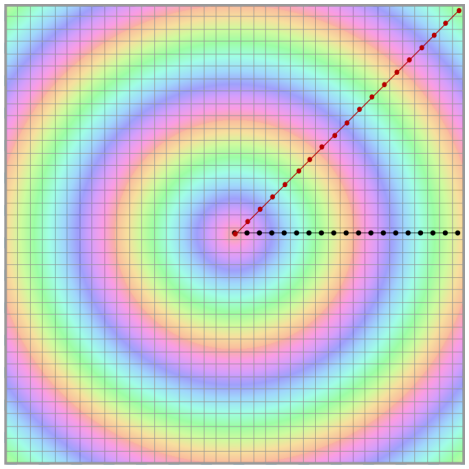
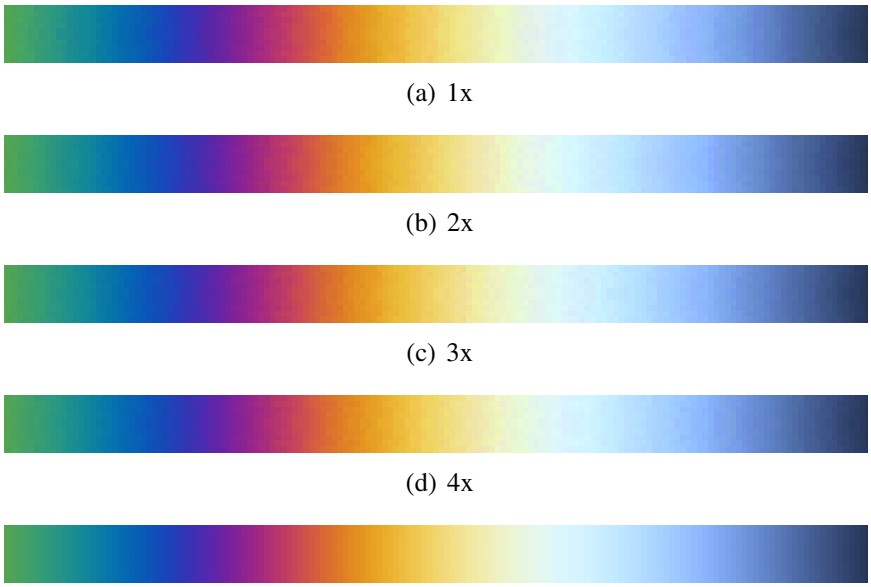


Figure 4.4: Pixel spacing along the diagonal is greater by a factor of $\sqrt{2}$.

We can instead use a new set of pixel locations spanning the half-diagonal distance but with finer spacing. To determine how fine that spacing should be, we need to recognize that we are, in effect, resampling the range of scattering angles. Sampling theory tells us that to correctly capture the highest-frequency features of the underlying data, we need to resample at least twice that frequency, the Nyquist limit. The worst mismatch between the original image resolution and the radial slice occurs when the slice is oriented at a 45˚ angle, along a diagonal of the image. In that case, the ratio of the sampling rates is $\sqrt{2}$ to 1. Therefore,

we need to increase the slice resolution by twice this ratio: $2 * \sqrt{2} \approx 2.8$. Rounding up, we use 3x sampling. Figure 4.5 illustrates the poor results produced by lower-resolution slices compared to higher resolutions and an image where each pixel is computed directly. Errors are particularly noticeable in the green/blue region of the 1x slice and the yellow/white region of the 2x slice.



(a) 1x

(b) 2x

(c) 3x

(d) 4x

(e) full-image (600 wvs/pixel, no sweep blending)

Figure 4.5: Effect of slice sampling rate (Slice Multiplier parameter)

4.2.3 Sweep shader

An OpenGL compute shader (`sweep.glsl`) generates a complete image from a radial slice. This shader operates on groups of pixels in the output image which share the same scattering angle in each of the four quadrants as shown in Figure 4.6. By simultaneously processing these groups of corresponding pixels, the number of calls to this shader is reduced by a factor of 4.

A further optimization based on the radial symmetry of the glory is possible. Only one quadrant of the final image needs to be generated. The final render to screen uses four quadrilaterals, each textured with the same quarter-image at the appropriate rotation.

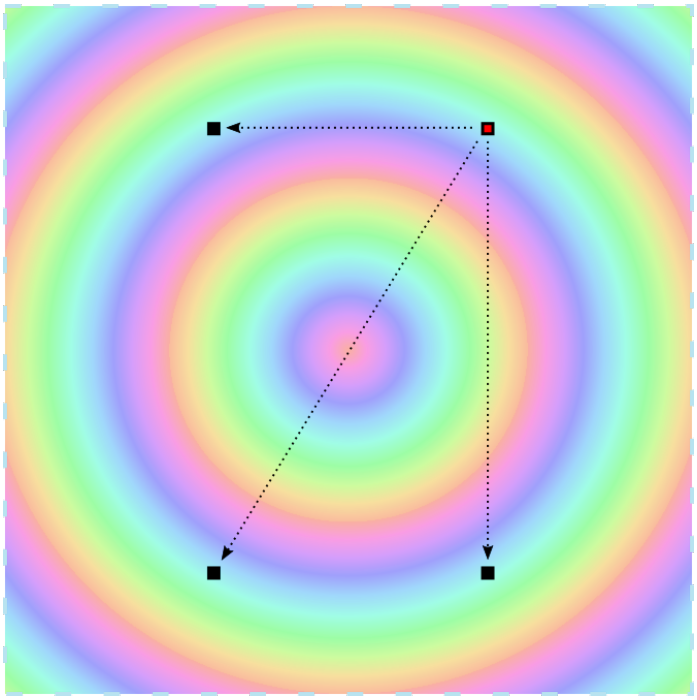


Figure 4.6: Leveraging the radial symmetry of the glory to reduce computation. The computed color for the red pixel in the first quadrant of the output image is copied to the corresponding pixels in the other three quadrants.

Because final image construction and rendering to screen consume such a trivial fraction of the total glory render time (less than 1 ms compared to hundreds of milliseconds for the scattering shader) I did not implement this additional optimization.

4.2.4 Blending

The shader selects and blends a user-specified number of samples around the pixel position to produce the final reported pixel value. Samples are drawn from a windowed Gaussian distribution over a circular region of radius 2 (shown in Figure 4.7). If the samples are located on a grid and the filter is not spatially varying, the 2D Gaussian is separable into two 1D passes. However, because the sweep shader is resampling from a radial slice being swept in an arc over the pixel grid, this method does not apply. Using a full 2D grid requires an impractical number of memory accesses for an $O(n^2)$ tap filter. With a filter radius of 2 image pixels, and a radial slice at 3x resolution, the resulting filter region is 12 slice pixels

Listing 4.1: Shader function using Box-Müller method for Gaussian distribution of sample points

```
vec2 GaussianDistrib(vec2 uu) {
  // smallest values of uu.x give largest values of radius
  // radius(0.213061) = 1
  // so redistribute [0,1) to [0.213061, 1)
  float minX = 0.213061;
  float x = uu.x * (1.0 - minX) + minX;

  float radius = sqrt(-2.0 * log(x*.5 + 0.5));
  float theta = M_PI_F * uu.y;
  return vec2(radius * cos(theta), radius * sin(theta));
}
```

wide, requiring 144 samples per filter. Limiting the filter to a 2-pixel-radius circular region reduces the count to $\pi r^2 = 36\pi = 113$ samples. A third option for spatially varying filters is to select randomized samples using importance sampling (Baek and Jacobs 2010), which increases noise but reduces the number of samples required. This is the method used for sampling and blending the radial slice pixels into final image pixels. It produces acceptable results with only 32 samples per filter instead of the 144 (or 113) required for a complete grid.

The shader selects samples using the method of Listing 4.1: A pair of pseudorandom values (u, v) over $[0, 1) \times [0, 1)$ is generated using 8-round Tiny Encryption Algorithm (see Section 5.1). u is scaled and shifted to sample from [0.213061, 1). This limits the resulting radius to (0,1] as shown in Figure 4.8. The Box-Müller method (Box and Müller 1958) uses (u, v) to importance-sample over a Gaussian probability distribution with $\sigma = 1$ to give angle and radius within a radius-1 circular sample region. The resulting radius is doubled so that samples are drawn from a two-unit-radius region, and finally (x, y) offset from the pixel location.

Adding the offset to the pixel coordinates gives the sample coordinates (s_x, s_y), which yield the sample's scattering radius $s_r = \sqrt{{s_x}^2 + {s_y}^2}$. Finally, $s_r * \mathit{SliceMultiplier}$ gives h_r, the index into the radial slice to find the corresponding slice pixel.

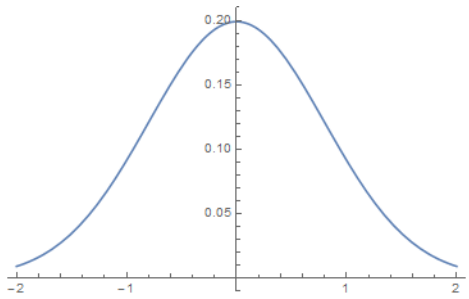


Figure 4.7: Windowed Gaussian distribution

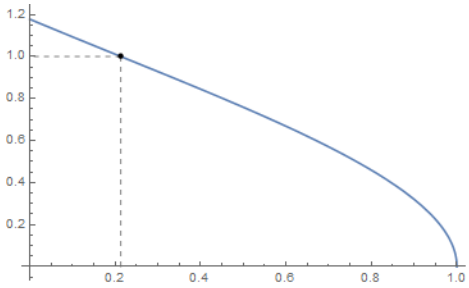


Figure 4.8: Gaussian radius function. $x = 0.213061$ is the minimum useful value, giving a radius of 1.0. Smaller values of x correspond to sample points outside the desired region.

4.2.5 Results: Total Mie calculations

Preliminary investigations demonstrated that, as expected, the Mie scattering stage of the glory-rendering pipeline was by far the most time-consuming stage of the entire process, with the wavelength-to-color conversion stage a distant second place. All the remaining stages, including sweeping and blending a radial slice into a full image, are nearly negligible. Therefore, reducing the time cost of the Mie scattering stage by reducing the number of Mie calculations can greatly speed up the overall process.

For a 512x512 image, a 4x resolution slice contains $ceil(512/2\sqrt{2} * 4) = 1449$ pixels. Compared to a complete image of 512×512 pixels, we obtain an ideal speedup factor for this stage of $512^2/1449 = 181$ from using a 4x radial slice.

4.2.6 Results: Timing

Table 4.2 and Figure 4.9 show timing comparisons for full-image generations and the radial slice method, using image sizes of 512x512 and 256x256 and slices at 1x, 2x, 3x, and 4x resolutions. These measurements were generated using pixel-based wavelength selection at 600 wavelengths per pixel. All times are reported in milliseconds. These are averages over five runs. For some parameter values, an initial startup delay was seen the first time the system was run with that configuration. These measurements were dropped and the average of the next five runs is given. Only the scatter and wavelength-to-color conversion shader phases timing results are presented as they consume the bulk of the computation time. Because sweeping and blending into a full image adds some overhead, we see less than the ideal speedup discussed earlier. However, even at 4x resolution, slice-based scattering calculations are enormously faster than full-image equivalent.

Image Resolution	Type & Multiplier	total wall-clock time	total wall-clock speedup	scatter GPU phase time	scatter GPU phase speedup	color GPU phase time	color GPU phase speedup
256	full	1913.61		1805.62		91.69	
256	slice 1x	131.01	15	107.78	17	6.87	13
256	slice 2x	240.76	8	210.90	9	13.89	7
256	slice 3x	347.22	6	308.07	6	20.69	4
256	slice 4x	445.43	4	402.31	4	23.60	4
512	full	7561.23		7188.20		363.43	
512	slice 1x	242.81	31	210.75	34	13.96	26
512	slice 2x	454.28	17	402.91	18	23.35	16
512	slice 3x	653.54	12	595.15	12	36.29	10
512	slice 4x	854.05	9	785.52	9	46.47	8

Table 4.2: Timing results (milliseconds) for full-image and radial slice at multiple slice resolutions.

4.2.7 Results: Image quality

Table 4.3 and Figure 4.10 show error comparisons for slices at 1x, 2x, 3x, and 4x resolution as compared to full-image generation. The slice-based images were generated using a two-

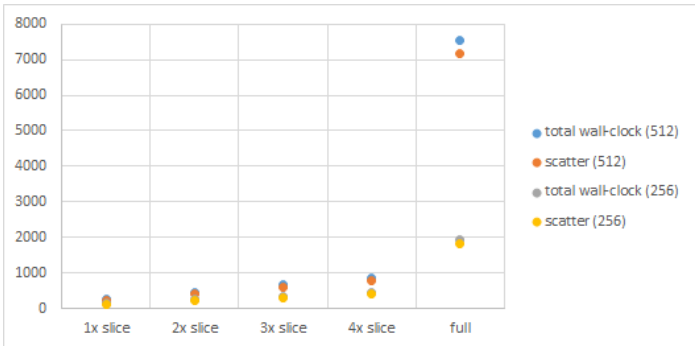
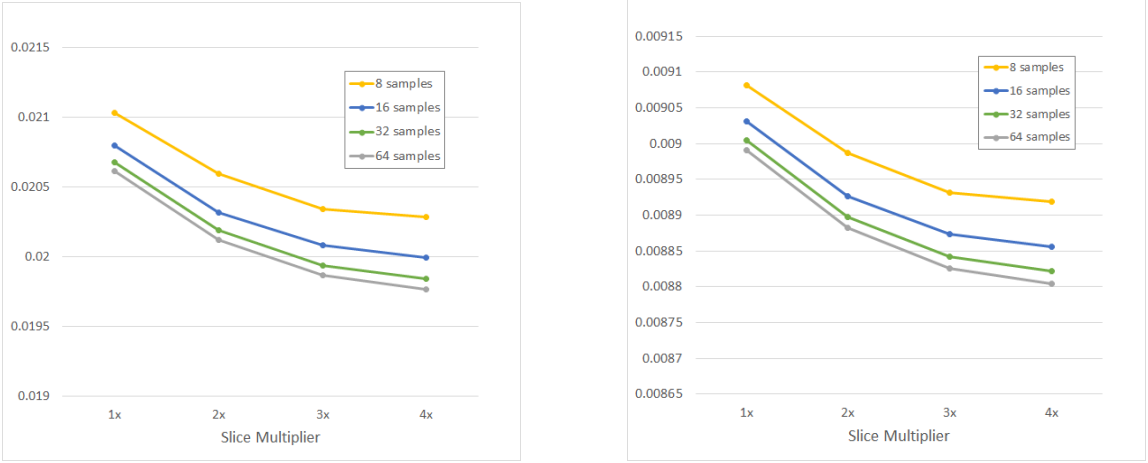


Figure 4.9: Timing results for slice vs full-image

pixel-radius Gaussian blend and four different sample group sizes (8, 16, 32, 64). All images were 512x512.

In all cases, the reduction in error from the 1x slice to the 4x slice is quite small. For ΔE^* error, the difference is only about 4%. The comparatively coarse 1x slice produces an image whose error is quite small, and may be sufficiently accurate for many graphics applications. The results presented here illustrate the tradeoff that can be made between render speed and render quality by changing the resolution of the radial slice.



(a) Average error

(b) RMS error

Figure 4.10: Error measurements for slice vs full-image, varying blend sample counts.

In addition to the perceptually-appropriate ΔE^* error metric, I compared the images using the pixel RGB values and using those values converted to LAB color space, as dis-

Slice mult	Blend samples	ΔE^* mean	ΔE^* RMS	RGB mean	RGB RMS	LAB mean	LAB RMS
1x	8	0.02103	0.00908	0.01748	0.00828	0.00638	0.00500
2x	8	0.02060	0.00899	0.01718	0.00821	0.00618	0.00492
3x	8	0.02034	0.00893	0.01719	0.00821	0.00605	0.00487
4x	8	0.02028	0.00892	0.01711	0.00819	0.00602	0.00486
1x	16	0.02080	0.00903	0.01733	0.00824	0.00628	0.00496
2x	16	0.02032	0.00893	0.01700	0.00817	0.00607	0.00488
3x	16	0.02008	0.00887	0.01704	0.00817	0.00594	0.00483
4x	16	0.02000	0.00886	0.01695	0.00815	0.00590	0.00481
1x	32	0.02068	0.00901	0.01724	0.00822	0.00624	0.00495
2x	32	0.02019	0.00890	0.01691	0.00814	0.00601	0.00485
3x	32	0.01994	0.00884	0.01695	0.00815	0.00588	0.00480
4x	32	0.01985	0.00882	0.01685	0.00813	0.00584	0.00479
1x	64	0.02061	0.00899	0.01719	0.00821	0.00621	0.00493
2x	64	0.02012	0.00888	0.01686	0.00813	0.00597	0.00480
3x	64	0.01987	0.00883	0.01690	0.00814	0.00584	0.00479
4x	64	0.01977	0.00880	0.01681	0.00812	0.00580	0.00480

Table 4.3: Differences from full-image render for radial slice renders at multiple slice resolutions and blend sample counts.

cussed in Section 3.3.2. The error images shown in Figure 4.11 have been brightened and contrast-enhanced for visibility. Figure 4.11(c) shows the slice image and the ΔE^* error image side-by-side for ease of comparison. Note that the bright bands in the error image, which indicate the highest errors, do not exhibit any clear correspondence to the color sequence of the rendered image.

4.3 Mie scattering algorithms and parameters

The visual scattering pattern of the physical glory phenomenon is fully described by the Mie scattering equations applied to all wavelengths over the continuous interval of the visible spectrum. A computer-generated rendering of a glory is produced by evaluating the Mie equations at a finite collection of discrete wavelengths. The image is very sensitive to the number and distribution of wavelengths used to calculate it. As shown by Laven (2003), 500-

(a) full-render image (b) slice image (c) comparison: slice and ΔE^*

(d) ΔE^* error (e) RGB error (f) LAB error

Figure 4.11: Final and error images for slice vs full-image. (Northeast quadrants)

600 evenly-spaced wavelengths are required for a reliable simulation. Laven illustrates the issue using MiePlot, which is a CPU implementation using uniform wavelength distribution.

The fundamental algorithm used for Mie scattering by most practitioners is BHMIE from Bohren and Huffman (1983). The original implementation is a serial CPU method developed in Fortran. Several later variations have added vector-structured optimizations (Warren J. Wiscombe 1979) and translated the BHMIE algorithm to other languages (for one collection, see SCATTERLIB, http://scatterlib.wikidot.com/mie). My research uses the basic Bohren-Huffman algorithm in a new OpenGL compute shader implementation restructured to take advantage of parallel GPU capabilities.

The main Mie-scattering function processes a single wavelength at a single scattering angle and returns the resulting intensity of that wavelength at that angle. To compute scattering results for multiple wavelengths and multiple scattering angles, it is called repeatedly.

4.3.1 Mie series expansion

Mie theory expresses the result as the sum of an infinite series of partial wave contributions. Early implementations continued generating series terms until the magnitude of the term coefficients was sufficiently small that the series was deemed to have converged, but this proved inadequate, as discussed in W. J. Wiscombe (1980). Wiscombe developed a formula which can be used *a priori* to determine the number of terms of the series expansion which will be required, as a function of x, the size parameter of the droplet. Recall from Section 2.2 that $x = \frac{2\pi r}{\lambda}$ where r is the radius of the droplet and λ the wavelength of the incident light. The stopping parameter is then $N = 4.05x^{1/3} + 2$. This formula for N is designed to fit or slightly overestimate the number of series terms required, and is valid when x is between 8 and 4200. Glories fall well within this range.

The main BHMIE function executes a large loop governed by this stopping parameter to produce the sequences of values $S1_n$ and $S2_n$, which are closely related to the scattering coefficients but can be computed more efficiently. At each step, this inner loop begins by generating A_n and B_n, the coefficients in the nth term of the Mie series. These coefficients are computed from the Riccati-Bessel functions X_n and ψ_n and the logarithmic derivative of ψ_n, denoted A_n. Next, A_n and B_n are combined with the angular functions π_n and τ_n, also generated through recurrence relations, to update $S1_n$ and $S2_n$. Finally, the Riccati-Bessel functions are updated by recurrence. At loop completion, the final complex scattering matrix coefficients $S1_N$ and $S2_N$ are returned to the calling function. For further details the reader is referred to the works cited above.

4.3.2 Upward and downward recurrence

There are two variations of the BHMIE main function, which differ in the treatment of A_n, the logarithmic derivative of ψ_n.

The first variation uses downward recurrence to compute A_n. That is, it starts with a value of n well in excess of N, the largest sequence index needed, and works downward,

computing and storing all values. This method is inefficient in both memory and computation, but guaranteed stable. The other uses upward recurrence to compute A_n. It starts with A_1, which depends only on x and m (defined in Section 2.2), and works upward, computing the next value only when needed by the main loop, and storing only the previous values needed for the next recursion. (Note that Fortran's 1-based loop indexing has been retained in later implementations.) This method is more efficient in both memory usage and runs much faster, but carries risks of numerical instability in the recursion relations and loss of accuracy due to subtraction of nearly-equal numbers (Warren J. Wiscombe 1979).

Warren J. Wiscombe (1979) developed an *a priori* method for determining from the size parameter and the complex index of refraction whether upward recurrence will fail in a particular Mie calculation, requiring use of the less- efficient downward recurrence method.

It was already known that upward recurrence may fail if (a) the imaginary part of the index of refraction is significantly greater than the real part, and (b) the size parameter is much greater than 1. The former condition indicates strong absorption, and the latter implies that many iterations of the recurrence relations will be executed, giving the numerical instability a chance to manifest. Wiscombe quantified the numerical relationships among these parameters which require use of downward recurrence. Additionally, Wiscombe determined that downward recurrence should always be used when $m_{Re} < 1$ or $m_{Re} > 10$ or $m_{Im} > 10$.

The refractive index of liquid water varies with the wavelength of incident light (Laven 2003). However, the imaginary part is negligible for wavelengths in the visible spectrum (The International Association for the Properties of Water and Steam 1997), so part (a) of the criterion does not hold for the atmospheric phenomena considered here. Additionally, for glories, coronas, and fogbows, size parameters are in the range 35 to 630. (These values correspond to a glory with $r = 4\mu$m droplets and $\lambda = 700nm$, and a fogbow with $r = 50\mu$m, $\lambda = 400nm$) Rainbows have size parameters of roughly 3000 to 15,000, and downward recurrence for ψ_n would be required for the upper end of this range. However, very large

rainbow droplets do not maintain a spherical shape as they fall, and so are out of the scope of this dissertation. Therefore, this work uses upward recurrence throughout.

Warren J. Wiscombe (1979) also addressed the question of upward vs. downward recurrence for ψ_n, a separate question from ψ_n's logarithmic derivative A_n. He determined that upward recurrence is almost as accurate as down-recurrence for $n < x$, is more efficient in both computation and storage, and is not subject to the overflow failure which ψ_n can exhibit in downward recurrence (ψ_n grows larger as downward recurrence proceeds). Therefore, the compute shader implementation uses upward recurrence to compute ψ_n. The second Riccati-Bessel function, χ_n, is always computed using upward recurrence, which is stable for this function.

4.3.3 Physical parameters

The only physical parameters which BHMIE exposes are x, the size parameter, and m, the complex refractive index of the droplet fluid.

In my glory-rendering test application, x is not directly set by the user. Instead it is derived from the droplet size, which is directly set by the user, and the light wavelength, which is programmatically determined based on the user's choices of wavelength count and sampling method. I make the simplifying assumption that the droplets are monodisperse; that is, they are the same size throughout the interaction region, as discussed in Section 2.1. Although this assumption is not physically valid for naturally-occuring glories, it is a reasonable simplification for research purposes to assess the benefits of the techniques presented in this dissertation.

I do not permit the user to set m. My system is hardcoded to use the index of refraction of water, which is real-valued. Furthermore, I make the simplifying assumption that the index of refraction is the same for all wavelengths of light considered by the glory calculations, as variation in this value is known to have little effect on the appearance of the glory (Laven 2008a). This fixed value allows the application to rely on the conclusions above regarding

recurrence method	image size	scatter GPU phase
upward	512	460.17
upward	256	235.38
downward	512	5790.68
downward	256	2987.98
approximation	512	20.17
approximation	256	10.15

Table 4.4: Timing results (milliseconds) for Mie scattering compute shader. Calculations were performed for 512x512 and 256x256 images with 2x radial slice.

stopping parameter and recurrence criteria.

4.3.4 Results

Table 4.4 and Figure 4.12 show timing results for the Mie scattering compute-shader phase of the glory rendering, using the upward- recurrence and downward-recurrence methods discussed in this section. The upward-recurrence method is faster by a factor of 12.694 for the 256x256 image, and 12.584 for the 512x512 image. To demonstrate how time-consuming even the faster upward-recurrence method is, timing data is also shown for an approximation function based on a previous high-resolution rendering (discussed in Section 7.2). Values are averages over five runs.

Table 4.5 shows rendered results and ΔE^* error images (compared to reference solution) for both recurrence directions. These images were generated using the WvPx sample selection method and 10 wavelengths per pixel. Error images have been brightened and contrast-enhanced for visibility. Although close inspection reveals small differences, these error images are nearly indistinguishable as compared to other error images in this chapter (such as Figure 4.11(d)).

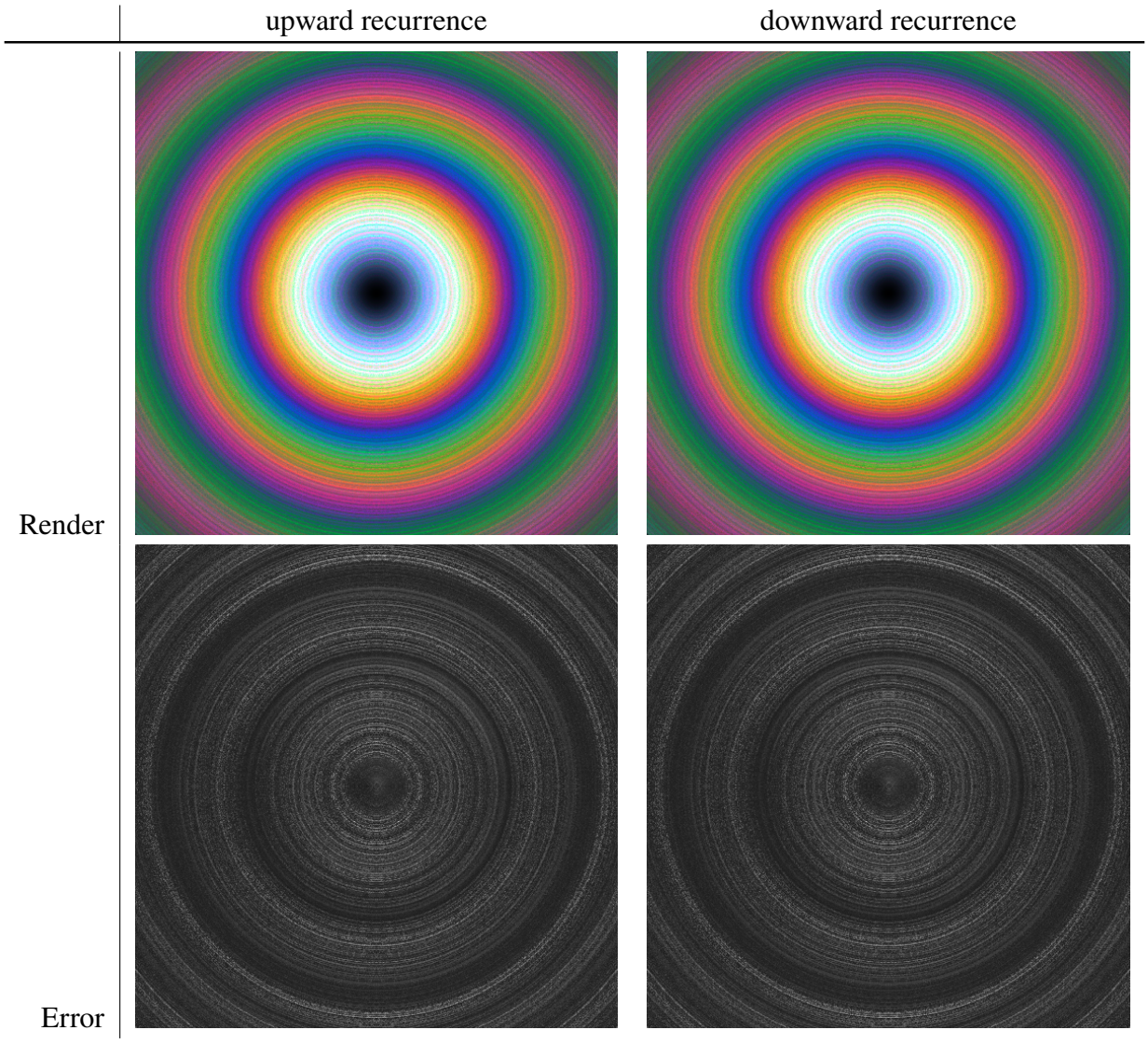


Table 4.5: Renders (top) and ΔE^* error images (bottom) for upward-recurrence vs. downward-recurrence Mie scattering algorithms.

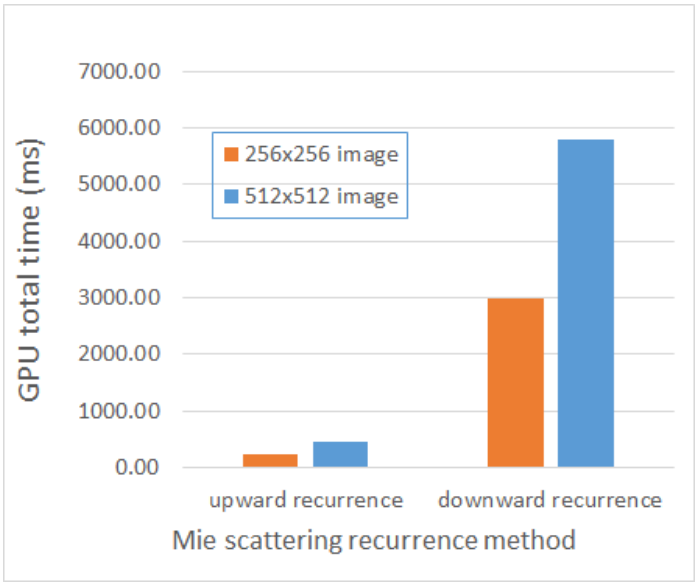


Figure 4.12: Comparison of upward-recurrence vs. downward-recurrence Mie scattering algorithm. Timing results are milliseconds of GPU time for scattering shader.

Chapter 5

SAMPLING METHODS

As discussed in Section 3.2.2, ComputeGlory contains a group of GLSL functions which determine the appropriate scattering angle θ and wavelength λ for a given shader invocation based on the index assigned to that invocation by the OpenGL system. These functions are called by multiple phases of the ComputeGlory compute shader pipeline.

I explored various organizational strategies to choose the (λ, θ) pairs for Mie scattering. These strategies are discussed in more detail in Chapter 6. In the present section, I explore a key piece of supporting functionality which all the pair-selection methods share: they require that the ComputeGlory system generate one or more sequences of values in the range [0.0, 1.0). Each sequence must be evenly distributed across that interval and share some of the statistical properties of a truly random sequence, but it must actually be deterministic and repeatable, so that multiple shaders can generate values from this sequence and be sure they are getting the same sequence of values with no coordination between shaders. Without this property, it would be necessary to pre-generate the sequence of values, store it, and have the host CPU program pass it to each shader that needs it. Although doing so would avoid repeated production of the same values, it would incur unacceptably high storage overhead for long sequences. Precomputing would also add to the overhead of calling each shader, as the stored data would have to be moved between CPU and GPU.

I investigated two different methods for generating the desired sequences.

5.1 Tiny Encryption Algorithm (TEA)

As its name implies, the Tiny Encryption Algorithm (TEA) was created as an encryption method (Wheeler and Needham 1995), designed to be extremely simple to implement. In

Listing 5.1: Tiny Encryption Algorithm core function.

```
1 uint2 TEA(uint2 v, uint4 key, int nRounds) {
2     unsigned int sum=0u;
3     for (int i=0; i<nRounds; ++i) {
4         sum += 0x9E3779B9u;
5         v.x += ((v.y+key.x)<<4u)^(v.y+sum)^((v.y+key.y)>>5u);
6         v.y += ((v.x+key.z)<<4u)^(v.x+sum)^((v.x+key.w)>>5u);
7     }
8     return v;
9 }
```

part due to its simplicity, TEA suffers from several cryptographic shortcomings, which are not relevant here; fortunately, the same simplicity makes it a useful algorithm to efficiently hash pixel positions to values on [0,1). Because TEA produces a pseudorandom sequence, not a quasirandom one, the output of TEA is not guaranteed to be evenly-distributed.

Listing 5.1 gives the core function of the TEA method as implemented in the `TEA.glsl` compute shader. It is a simple block cipher operating on two 32-bit values using a 128-bit key. The hard-coded constant added inside the TEA loop (line 4) is a key-schedule constant used to prevent simple symmetry-based attacks when TEA is used for encryption. Its value is defined by the TEA authors to be $\lfloor 2^{32}/\phi \rfloor$, where ϕ is the golden ratio.

The Tiny Encryption Algorithm has the known issue that it must be run through several rounds of processing to get evenly-distributed results. Figure 5.1 shows how the sample points form clumps and gaps when only two rounds of processing are used, while these artifacts diminish with 4 and 8 rounds. The number of rounds required for a particular application depends on how the sample values are used.

5.1.1 Tiny Encryption Algorithm (TEA) implementation and usage

An overview of the Tiny Encryption Algorithm and its properties was given in Section 2.10. Zafar, Olano, and Curtis (2010) demonstrated that TEA is a suitable choice as a GPU pseudorandom number generator, as it provides repeatability, random access, multiple independent streams, and speed, and its output is free from detectable statistical bias. Additionally, it uses only simple operations and requires very little internal state or data. Finally, it can

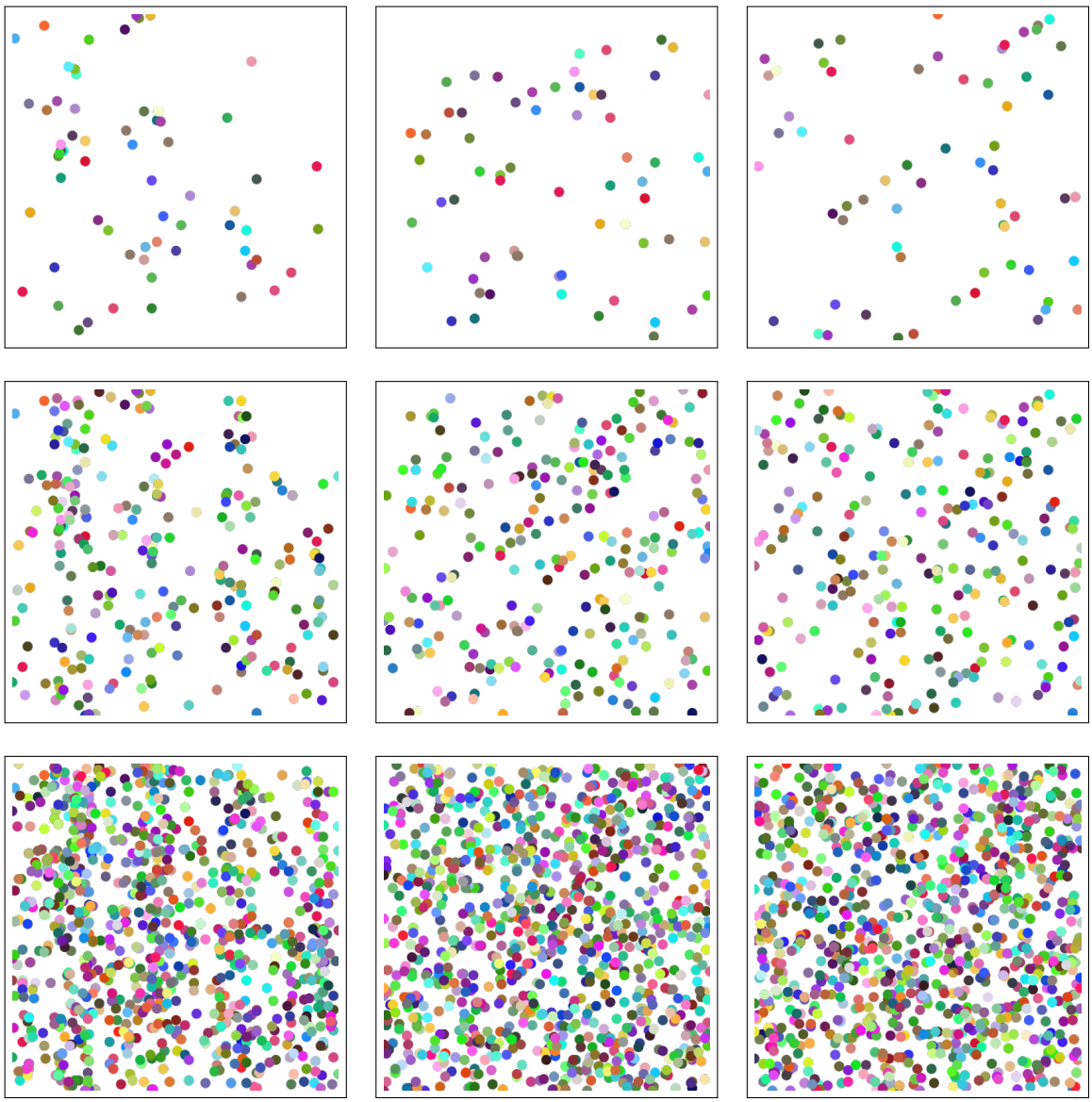


Figure 5.1: TEA samples with 2, 4, and 8 rounds of processing. Top row: 64 samples. Middle row: 256 samples. Bottom row: 1024 samples.

Listing 5.2: Using the Tiny Encryption Algorithm (TEA) to generate pseudorandom values in [0.0, 1.0).

```
1  if (DoSlice == 1)
2  {
3    // derive x,y as if the slice buffer dimensions
4    // were sliceMultiplier x N
5    // instead of 1 x (sliceMultiplier*N)
6    uint D = ImageSize / SliceMultiplier;
7    uxy = uvec2( id.x / D, id.x \% D);
8  }
9  else
10   uxy = id.xy;
11
12 uvec2 tea = TEA(uxy, uvec4(0), 8);
13 tea.x += SampleIndex + 1;
14 tea.y += SampleIndex + 1;
15 tea = TEA(tea, uvec4(0), 8);
16
17 return ( vec2(float(tea.x), float(tea.y)) / float(UINT_MAX) );
```

easily be adjusted to achieve a desired tradeoff between speed and quality.

Listing 5.2 shows how TEA is employed by the ComputeGlory shaders. The first step is to construct appropriate input values. The TEA function takes a pair of unsigned integers as its input vector, and returns a new pair of unsigned integers. When performing a full-image glory rendering, the two-dimensional shader invocation index is used as the initial vector (line 10). Each shader invocation queries the OpenGL compute shader dispatch system to obtain its index. When using the radial slice optimization discussed in Section 4.2, the 1D invocation index is reinterpreted as though the slice were a 2D array whose dimensions are the slice resolution multiplier and the base pixel count of a single-resolution slice (lines 6-7). That is, we assemble v from the input x by separating x into quotient and modulus relative to the size of the single-resolution image.

TEA also requires a 4-vector of unsigned integers as the encryption key. Since we are not concerned here with encryption, merely using TEA as a pseudorandom number generator, the choice of key is not important. I use $(0, 0, 0, 0)$ for simplicity.

The third input to TEA is the number of rounds of encryption to be run. Lines 12 through 15 run 8 rounds using *uxy* as the initial input vector, then add SampleIndex, which

is a system-wide unique value identifying the (λ, θ) pair being generated, and run another 8 rounds.

Line 17 converts the final pair of unsigned integers to floats in [0.0, 1.0).

The configuration of TEA used in this example, with 8 encryption rounds, addition of the index, and 8 more rounds (abbreviated 8/8) was determined empirically for this application. More rounds of TEA, whether before or after adding the index, gave no further improvement in final image quality. Many other TEA configurations are possible, such as encoding the index into the initial key and performing only one group of TEA iterations, performing more or fewer total rounds, and dividing the rounds unevenly (e.g. 4/8).

There are also many other hash functions and cryptographic algorithms that meet the criteria given by Zafar, Olano, and Curtis (2010), differing in performance and ease of implementation. For the present discussion, the particular random number generator and its configuration details are not important so long as the results meet the criteria for a good pseudorandom number generator. Therefore, once I established that 8/8 TEA was satisfactory I did not explore further pseudorandom methods.

5.2 Sobol sequence

The Sobol sequence (Sobol' 1976) is a popular globally-coherent low-discrepancy sequence. A given sequence is determined by its *direction numbers*, which are a specially-derived set of binary values. Most applications use the optimized direction numbers published by Joe and Kuo (2008). To produce a Sobol sequence of length n requires one direction number for each bit in the binary expansion of n per dimension of the output, a total of $\lceil Log_2(n) \rceil$ direction numbers for each dimension.

The sequence values can be generated independently and in parallel – that is, there is no need to generate the first $n - 1$ values of the sequence before generating the n^{th}. We simply need the sequence index of the desired value.

5.2.1 Sobol sequence generating algorithm

One version of the classic algorithm to generate a Sobol number in one dimension is given in Listing 5.3. Each of the direction numbers is XOR'd into the result if the corresponding bit in the index is set.

When this algorithm is applied to generate multiple samples per pixel of an image, a modification is commonly applied to avoid unwanted correlation between pixels. Line 1 of Listing 5.3 is replaced with a random number generated from a hash of the pixel position (Kollig and Keller 2002). This has the result of shuffling the samples from one pixel to the next.

Listing 5.3: Algorithm to generate the n^{th} Sobol number from its index and a set of direction numbers, `DirNum`

```
1 result = 0;
2 for(i=0; i < maxBits; ++i)
3   if (index & (1 << i) != 0)
4     result ^= DirNum[i]
```

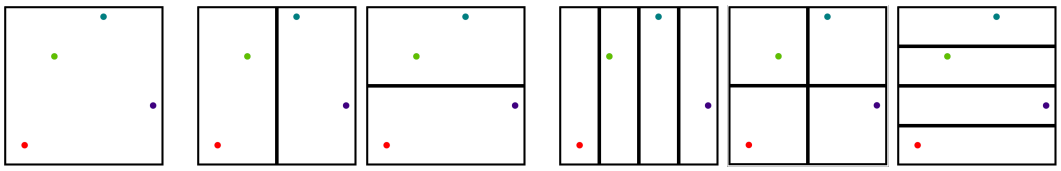
5.2.2 Sobol matrix formulation

The algorithm in Listing 5.3 can be expressed as a binary matrix-vector multiplication (Bratley and Fox 1988). In this form, the index is a column vector of bits, most significant first. Each direction number becomes a column of the matrix, with the direction number for the most significant bit of the index on the left, and the least significant on the right. The product can be evaluated using binary arithmetic, with AND for multiply and XOR for add, or (equivalently for integers) using modulo 2 arithmetic. Equation 5.1 shows how to produce a 5-bit Sobol value from a 5-bit index, using the direction numbers for dimension 3 provided by Joe and Kuo (2008).

$$\begin{pmatrix} v \\ a \\ l \\ u \\ e \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} \mod 2. \tag{5.1}$$

5.2.3 The (t, m, s)-net property

The (t, m, s)-net property describes the uniformity of point sets by considering the distribution of the points into cells known as *elementary intervals*. In base $b = 2$, a point set is a (t, m, s) net if, for 2^m points in s dimensions, there are 2^t points in every 2^{t-m}-sized elementary interval. Figure 5.2 shows this in 2D ($s = 2$), with four points ($m = 2$). Figure 5.2(a) is the $(2, 2, 2)$ case: there are 2^2 points in an interval of size 2^0 (1×1); (b) is the $(1, 2, 2)$ case: there are 2^1 points in each interval of size 2^{-1} (0.5×1 or 1×0.5); finally (c) is the $(0, 2, 2)$ case: there are 2^0 points in each interval of size 2^{-2} (0.25×1, 0.5×0.5, or 1×0.25).



(a) t=2 (b) t=1 (c) t=0

Figure 5.2: The (t, m, s)-net property for four points in 2D with $m = 2$ and $s = 2$.

The Sobol sequence is a (t, s) sequence, meaning defined subsets of the points in the sequence form a (t, m, s) net. Joe and Kuo (2008) derived their set of "good" direction numbers by optimizing for this property over m and t. In an image, each pixel is an elementary interval, so by choosing an appropriate m and pair of cell index dimensions X,Y the Sobol sequence will place the same power-of-two number of samples in every pixel. In particular, for a number of samples equal to the number of pixels, we know there is one and only one unique sample location per pixel. For one Sobol sample per pixel, just as the index is sufficient to uniquely identify the pixel, the pixel location is enough to uniquely identify the

point index. In other words, the matrices which convert Sobol sequence index to sample locations are invertible in this case.

5.3 Cell-Indexed Sobol

A recently-developed method for generating and applying the Sobol sequence, called *Cell-Indexed Sobol* (CI-Sobol), can be used to minimize the discrepancy in the sample distribution between adjacent cells as well as within a single cell (Olano and Blenkhorn, in preparation). CI-Sobol expands on the Sobol matrix partitioning technique introduced in Grünschloß, Raab, and Keller (2012). Rather than drawing samples evenly distributed in n dimensions for each cell, CI-Sobol includes the k-dimensional cell coordinates in the low-discrepancy distribution, and draws its samples from $n + k$ dimensions. These additional dimensions are called the *index dimensions*. For samples in image pixels, the index dimensions correspond to the (x, y) pixel coordinates and the samples are drawn from $n+2$ dimensions of the Sobol sequence.

Cell-Indexed Sobol enables rapid generation of values of the Sobol sequence based on their location in a multi-dimensional grid (the index dimensions), rather than by their index within the Sobol sequence. Points drawn by cell or pixel in this manner are called *cell indexed* samples. Using CI-Sobol, an algorithm can easily loop over the pixels of an image *in pixel order*, generating the appropriate Sobol values for each pixel and performing whatever processing is desired. Depending on the application, this may give significant locality and caching benefits compared to accessing pixels in the order dictated by the Sobol sequence.

CI-Sobol partitions and recombines the Sobol direction-number matrices (Section 5.2.2) in an approach which extends the work of Grünschloß, Raab, and Keller (2012). It produces an alternate set of direction numbers that enable cell-indexed access to the sample points in the index dimensions as well as any additional Sobol dimensions desired.

Both the earlier work and the CI-Sobol method rely on the fact that the matrix in

Equation 5.1 is guaranteed to be non-singular, since it produces a unique value for each index (Bratley and Fox 1988), and therefore it is invertible. This means that, given a Sobol sample value, it is possible to get the Sobol sequence index back from that value:

$$\begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} v \\ a \\ l \\ u \\ e \end{pmatrix} \quad \text{mod } 2. \tag{5.2}$$

Consider a 5-bit example, where two samples are being placed in each cell of a 4x4 grid. This can be accomplished by using two dimensions of a Sobol series to give X and Y coordinates of each sample. For each dimension, we need a 5-bit value, in which the most significant two bits identify the cell, and the least significant three bits give the sub-cell location. Note that two different Sobol matrices are required, constructed from the direction numbers for the corresponding Sobol dimensions. Equation 5.3 shows the matrix for the Y dimension. The first two rows of the matrix (filled green) produce the Y cell coordinate (YY) from the index, and the bottom three (green outline) produce the Y component of the sub-cell position (yyy):

$$\begin{pmatrix} Y \\ Y \\ y \\ y \\ y \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} \quad \text{mod } 2. \tag{5.3}$$

Similarly, Equation 5.4 shows the matrix for X partitioned into the rows which produce the X cell coordinate (filled red) and the rows which produce the sub-cell location (red

outline):

$$\begin{pmatrix} X \\ X \\ x \\ x \\ x \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} \quad \text{mod } 2. \tag{5.4}$$

Recall each group of 2^m indices generates one sample for each of the 2^m cells, as discussed in Section 5.2.3. This means the most significant $n - m$ bits of the index enumerate the sample number within the cell and the lower m bits enumerate the cells.

$$index = (\underbrace{i_1 \ \dots \ i_{n-m}}_{sampleID} \ \underbrace{i_{n-m+1} \ \dots \ i_n}_{cellID}) \tag{5.5}$$

Continuing the 5-bit example with 16 cells, we have $m = 4$ and $n - m = 1$, so the least significant four bits of the index enumerate the 16 cells and the top bit enumerates the within- cell samples, i.e. whether a given index corresponds to the first sample for a cell or the second.

Figure 5.3 shows the relationship between sequence index and the placement of samples in cells. Note that in this example, the first 16 indices (red) place a single point in each

Figure 5.3: Point order for 4x4-cell example

cell, and the next 16 indices (blue) place a second point in each cell. Note that the cells are not enumerated in the same order for $S = 0$ and $S = 1$. Putting this in the form used for X and Y shows that an identity matrix transforms input (index) into output (also index). The top row of that matrix (filled blue) is responsible for producing the sample number S from the index, and the bottom rows (blue outline) for extracting the bits that give the cell location.

$$\begin{pmatrix} S \\ s \\ s \\ s \\ s \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} \quad \text{mod } 2. \tag{5.6}$$

These three partitions are enough to uniquely identify the sample number and X and Y cell coordinates from the index. This partitioning approach was introduced by Grünschloß, Raab, and Keller (2012) to allow adaptive sample densities per cell of a Sobol or Halton distribution in quasi-Monte Carlo applications.

$$\begin{pmatrix} S \\ Y \\ Y \\ X \\ X \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} \quad \text{mod } 2. \tag{5.7}$$

The number of bits was chosen so that there is a unique mapping between sequence indices and (S, Y, X) triplets. Therefore, this matrix must be invertible, resulting in a new matrix of direction numbers that reproduces the index from the sample number and cell

coordinates. For the matrix in (5.7), the result is

$$\begin{pmatrix} i \\ n \\ d \\ e \\ x \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} S \\ Y \\ Y \\ X \\ X \end{pmatrix} \mod 2 \tag{5.8}$$

Switching to more compact notation, this is

$$i = M_{i\to Q}^{-1} Q = M_{Q\to i} Q \mod 2. \tag{5.9}$$

5.3.1 Combining direction-number matrices

While being able to invert the matrices to produce an index from the cell coordinate is useful, in some cases, the number of cells (and index size) necessary to construct an invertible matrix can be large. The largest for pairs of dimensions from the Joe and Kuo (2008) set is 132 for dimensions 466 and 9540. With 66 bits each of X and Y cell coordinates, and 132-bit direction numbers, we can derive a 132-bit index for any cell. Both of these are larger than the word sizes of modern computers, so directly generating the index through the inverse is often impractical. Fortunately, this is unnecessary.

Since Equation (5.9) produces the original index, we can combine it using mod-2 matrix multiplication with the regular (forward) Sobol matrix for any other Sobol dimension as shown in Equation (5.10).

$$\begin{aligned} K &= M_{i\to k}\, i \mod 2 \\ &= M_{i\to k}\, M_{Q\to i}\, Q \mod 2 \\ &= M_{Q\to k}\, Q \mod 2. \end{aligned} \tag{5.10}$$

Here K is the value of the sample point with the specified index from Sobol dimension k, $M_{i\to Q}^{-1}$ is the matrix in Equation (5.9), and $M_{i\to k}$ is the binary direction-number matrix

Listing 5.4: Algorithm to generate a Sobol number from pixel X, pixel Y, and sample number given the direction numbers for the new dimension, `NewNum`.

```
result = 0;
for(i=0; i < XPixelBits; ++i)
  if (XPixel & (1 << i) != 0)
    result ^= NewNum[i]
for(i=0; i < YPixelBits; ++i)
  if (YPixel & (1 << i) != 0)
    result ^= NewNum[XPixelBits+i]
for(i=0; i < SampleBits; ++i)
  if (Sample & (1 << i) != 0)
    result ^= NewNum[XPixelBits+YPixelBits+i]
```

Listing 5.5: 32-bit Cell-Indexed Sobol generator

```
vec2 getSobol32(uvec2 uxy, uint SampleIndex)
{
  uint s = (uxy.x<<21 & 0xffe00000u) | (uxy.y<<10 & 0x001ffc00u);
  s = s | (SampleIndex & 0x3ffu) ;
  uint state = 0u;
  for(int i=0; i<32; ++i, s>>=1u)
        state ^= sobol32[i] * (s & 1u);

  // Separate 32-bit state into
  // 16-bit X (top half) and Y (lower half)
  uint fx = (state>>16) & UINT_16BIT_MAX;
  uint fy = state & UINT_16BIT_MAX;

  // divide by 16-bit max to produce values in [0,1)
  return ( vec2( float(fx), float(fy) )  / float(UINT_16BIT_MAX) );
}
```

that generates sample values from dimension k. The resulting matrix $M_{Q\to k}$ contains a new set of direction numbers to directly generate complete samples for dimension k from the pixel coordinates and per-pixel sample number. These can be applied using a simple modification to the standard algorithm, with minimal change to any existing code already using Sobol for quasirandom number generation (Listing 5.4).

5.3.2 CI-Sobol implementation and usage

Listing 5.5 shows the GLSL function which, given cell coordinates and a within-cell sample index, produces corresponding 16-bit values from dimensions 2 and 3 of the Sobol sequence. Lines 3-4 construct a 32-bit Sobol index from cell coordinates (11 bits each, enough to

specify pixels in a 2048x2048 image) and within-cell sample index (10 bits, enough to specify up to 1024 wavelengths per pixel, well above the theoretical minimum 500-600). Lines 6-7 perform the binary operations to generate a 32-bit value containing both output samples. Lines 11-12 separate the 32-bit Sobol output into the desired 16-bit segments. Finally, line 15 converts these values to floats in $[0, 1)$.

5.4 Comparison of distribution quality for TEA vs Sobol

The Tiny Encryption Algorithm has the known issue that it must be run through several rounds of processing in order to get evenly distributed results. Table 5.1 shows how the sample points cluster when only two rounds of processing are used, forming clumps and gaps (top row). These artifacts diminish if the TEA processing is increased to 4 and 8 rounds (middle and bottom rows). The number of rounds required for a particular application depends on how the sample values are used. Figure 5.4 shows visual artifacts (streaks and grainy noise) resulting from glory renderings using too few rounds of TEA to select the wavelengths scattered for each pixel. These images were produced with 2 (left) and 4 (right) rounds of TEA before adding the SampleIndex, and 2 rounds afterwards (lines 12-15 of Listing 5.2). These images have been brightened and contrast-enhanced for visibility.

The Sobol sequence, on the other hand, was designed to produce an evenly-distributed set of samples, and to do so (to the extent possible) even when the total sample count is small, as illustrated in Figure 5.5.

Because so many rounds of TEA are needed, in practice TEA may require more calculations than Sobol. The tradeoff is that the TEA calculations are simple and straightforward compared to Sobol. Both produce deterministic, repeatable sequences.

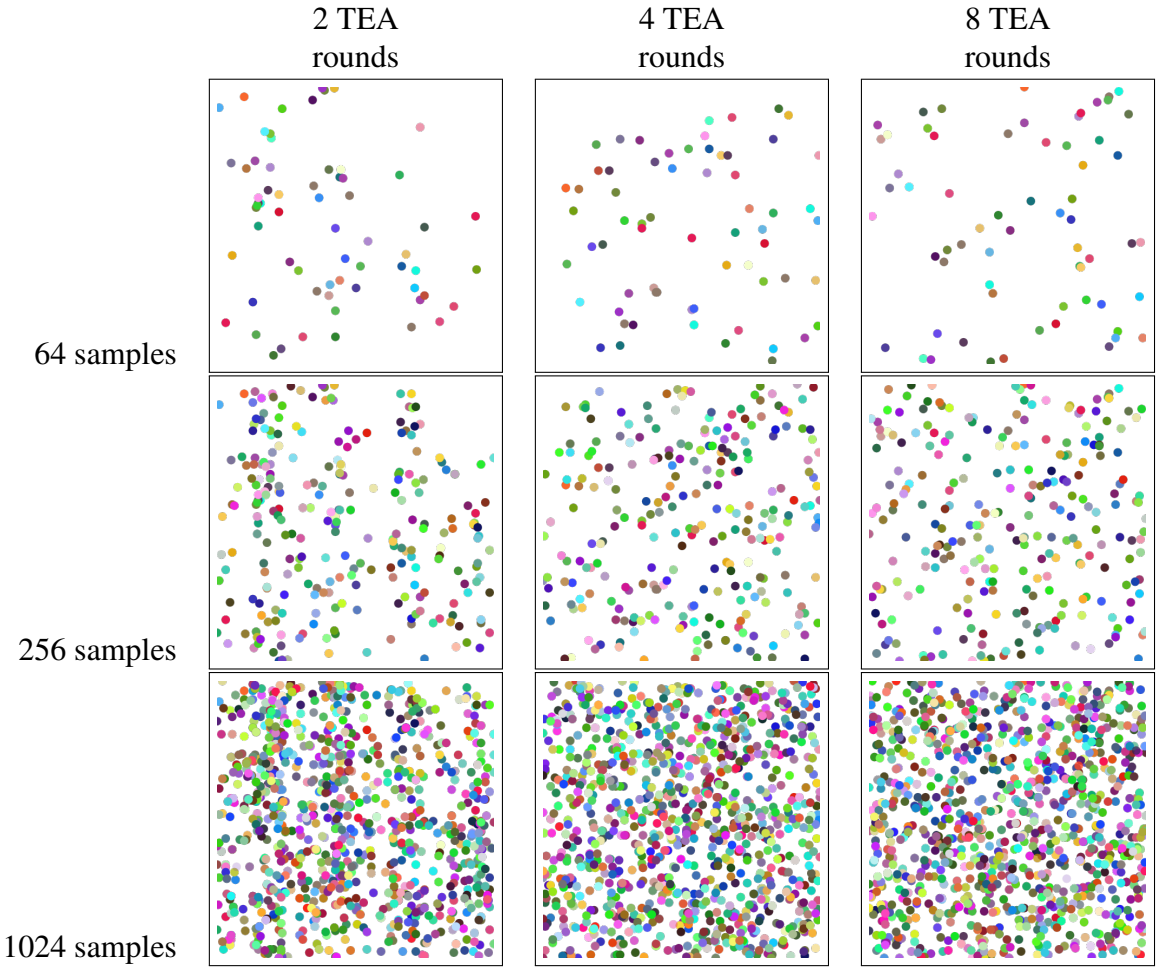


Table 5.1: TEA sample sequences with 2, 4, and 8 rounds of processing.

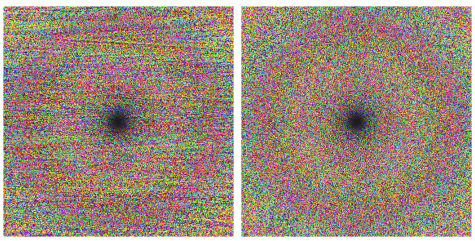


Figure 5.4: Visual artifacts in glory renderings using TEA with insufficient rounds of processing.

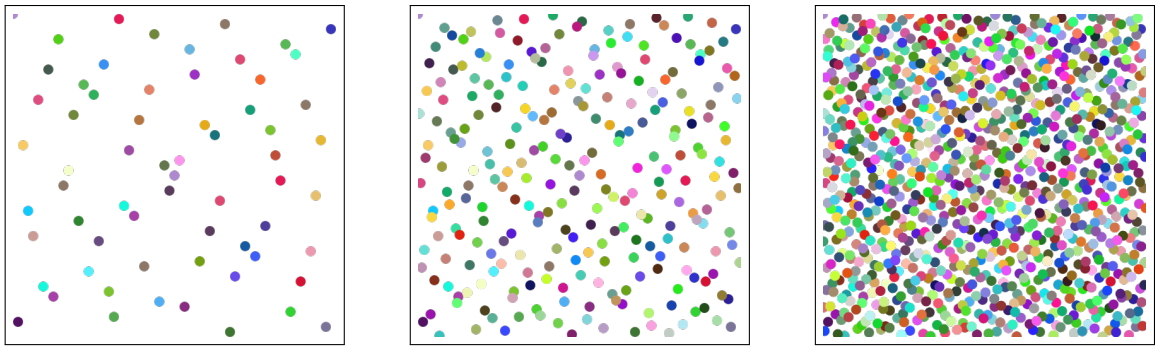


Figure 5.5: Sobol sequence with 64, 256, and 1024 samples.

5.5 Sobol sequence implementation: direct calculation vs. lookup tables

The implementations of TEA and Sobol sequence generation discussed in the previous sections of this chapter are both direct calculation, meaning that sequence values are generated on the fly and returned to the caller. Results are not cached, so exactly the same calculations are performed again when later compute shaders process the same (λ, θ) pair. Although this causes redundant computation, the inefficiency is preferred over precomputing, storing, manipulating, and accessing large arrays of sequence values.

However, Sobol sequence generation offers a compromise between direct calculation and precomputation of the entire sequence. Both forward-Sobol and CI-Sobol use only the binary XOR and AND operations to produce sequence values from their inputs. These operations are associative and commutative. By reordering and regrouping these operations, it is possible to separate the bits of a large sequence index into several smaller chunks, which can be processed separately in any order, then recombine the partial results to yield exactly the same sequence value as would be produced by processing the entire index.

Furthermore, the results of processing these smaller sub-indices against the Sobol matrix can be pre-computed offline and stored as lookup tables, avoiding redundant calculation in situations where we compute many Sobol values. Using lookup tables reduces runtime sequence access to a small number of table lookups and bitwise operations, at the cost of a small, fixed amount of advance computation to generate the lookup tables, and storage for

Listing 5.6: Using the Sobol sequence to generate a pair of quasirandom values in [0.0, 1.0).

```
1  uint SampleIndex = SobolSampleBase + id.x;
2  // initialize accumulators for Sobol dimensions 2 and 3
3  uint result2 = 0;
4  uint result3 = 0;
5  for (uint i = 0; SampleIndex != 0; SampleIndex >>= 1, ++i)
6  {
7    // is current lowest bit set?
8    if (SampleIndex & 1) {
9      result2 ^= sobol2[i];
10     result3 ^= sobol3[i];
11   }
12 }
13 return ( vec2(float(result2), float(result3)) / float(UINT_MAX) );
```

those tables. In an interactive application, incurring one-time offline setup costs in order to accelerate runtime access is an entirely appropriate tradeoff.

I wrote a separate Python script (`sobol_tables_vs_direct.py`) which generates tables at specified index sizes and sub-table division schemes, and exhaustively compares Sobol values generated by those tables against values generated directly from the full index to ensure correctness of the sub-table data and its usage.

Listing 5.6 shows the GLSL function which performs direct Sobol computation, found in `wavelength.glsl`. Given a sequence index, the function calculates and returns the appropriate sample values from two separate Sobol dimensions. This code is modified from Grünschloß, Raab, and Keller's implementation of the algorithm published by Joe and Kuo (2008) and uses the direction numbers from the earlier work.

Line 1 constructs a Sobol sequence index for the desired values. The base sequence index of the current sample cohort is a parameter passed to the shader by the host program. The OpenGL invocation index (`id.x`) is treated as an index into the current sample cohort. Adding these gives the actual Sobol index.

The loop spanning lines 5-12 runs across the bits of the sequence index in turn. Line 8 checks whether the current bit is set. If so, lines 9 and 10 retrieve the direction numbers for the current bit from dimensions 2 and 3 and XOR these values into the accumulators.

Next, consider the GLSL function shown in Listing 5.7, which performs table-based

Listing 5.7: Using the Sobol sequence to generate a pair of quasirandom values in [0.0, 1.0).

```
1   uint SampleIndex = SobolSampleBase + id.x;
2   uint numSobolSegments = 4;
3   uint bitsPerSegment  = 5;
4   uint entriesPerSegment = uint(pow(2, bitsPerSegment));
5   uint mask = 0x1f;  // 5 bits, all 1s
6
7   // initialize accumulators for Sobol dimensions 2 and 3
8   uint result2 = 0;
9   uint result3 = 0;
10
11  // for each segment of the index
12  for (uint seg = 0; seg < numSobolSegments; seg++)
13  {
14    // extract the corresponding bits of the index
15    uint bits = ( SampleIndex >> (seg * bitsPerSegment) ) & mask;
16    // and use those to index into the appropriate table
17    // [region of shaderbuffer]
18    uint i = seg*entriesPerSegment + bits;
19    result2 ^= sobolTable2Buf[i];
20    result3 ^= sobolTable3Buf[i];
21  }
22
23  return ( vec2(float(result2), float(result3)) / float(UINT_MAX) );
24 }
```

Sobol computation. Like the direct-calculation version above, this is found in `wavelength.glsl`. Given a sequence index, this function calculates and returns the appropriate sample values from two separate Sobol dimensions, producing exactly the same output as the direct calculation function in Listing 5.6. In this example, a 20-bit sequence index is broken into four five-bit segments. (The choice of table subdivision scheme is discussed in Section 5.5.1.) Each index segment maps to 32 different outputs. The four 32-entry lookup tables have been previously computed off-line.

Line 1 constructs the Sobol index in the same manner as in Listing 5.6. The loop spanning lines 12-21 runs across the segments in turn. Line 15 shifts and masks the index to extract the bits of the current segment, producing a five-bit value. This value is the index into the corresponding lookup table for each dimension. In the ComputeGlory implementation, the tables for a given Sobol dimension are passed from the host application to the GLSL function in a single shaderbuffer, so line 18 computes the index into the shaderbuffer for segment `seg` and table index `bits`. The corresponding values are retrieved from the

dimension 2 and dimension 3 shaderbuffers and XORed with the accumulating results in lines 19 and 20. Finally, line 23 normalizes the results to [0,1).

5.5.1 Comparison of table-lookup implementations

The table-lookup formulation of the Sobol sequence led to two implementation questions, possibly interdependent:

1. **Partial-table bit size**. Based on the typical image size (1024x1024), radial slice resolution, and expected maximum number of Mie calculations needed, a 20-bit index is sufficient. An index of this length can be subdivided into 20 1-bit tables, 10 2-bit tables, 5 4-bit tables, and so on. What bit size produces Sobol sequence values fastest? Or is table configuration not important?

2. **Table storage and management**. The Sobol sequence lookup tables can be stored on the GPU as hardcoded tables in the compute shader, or on the CPU, using shader data buffers to pass the tables to the GPU from the host application. While the former minimizes the data-handling overhead and complexity, the latter minimizes the shader code size and the shader's data storage footprint. Which table storage scheme produces Sobol sequence values fastest? Or is storage scheme not important?

It was not initially clear whether the two table-based implementation questions were independent or linked – that is, whether the optimal table subdivision scheme differed with table storage method. Therefore, I performed a three-way comparison of direct calculation, shaderbuffers, and in-shader tables at all table bit sizes that were supported by my system configuration. The in-shader tables were limited to 1,2,4, and 5-bit options as 10-bit tables (2 tables x 1024 entries x 32 bits) were too large for the shader code processing engine.

Figure 5.6 shows the result of this comparison. Reported times are the total GPU time consumed by the Mie calculation, color conversion, and merge phases of my pipeline, as these are the compute shaders which perform large numbers Sobol calculations and repeat

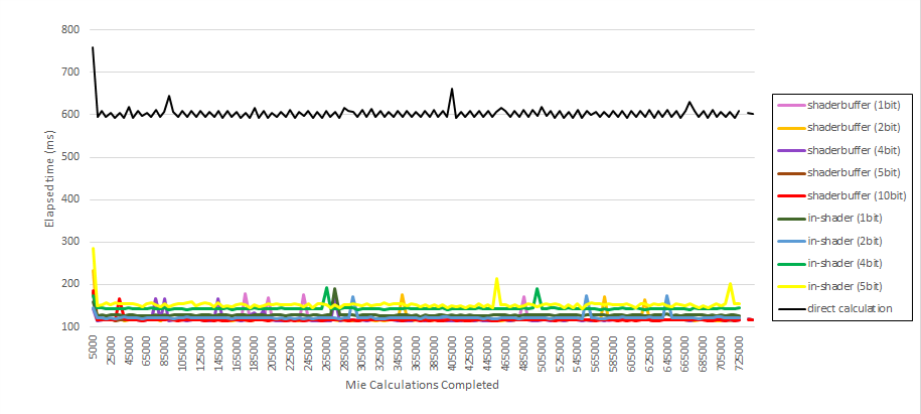
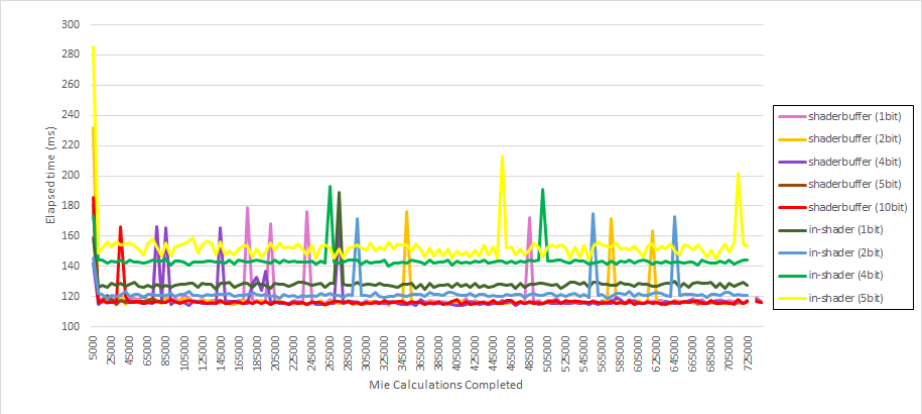


Figure 5.6: Timing results (ms) for direct Sobol sequence calculation, shaderbuffer tables, and in-shader tables.

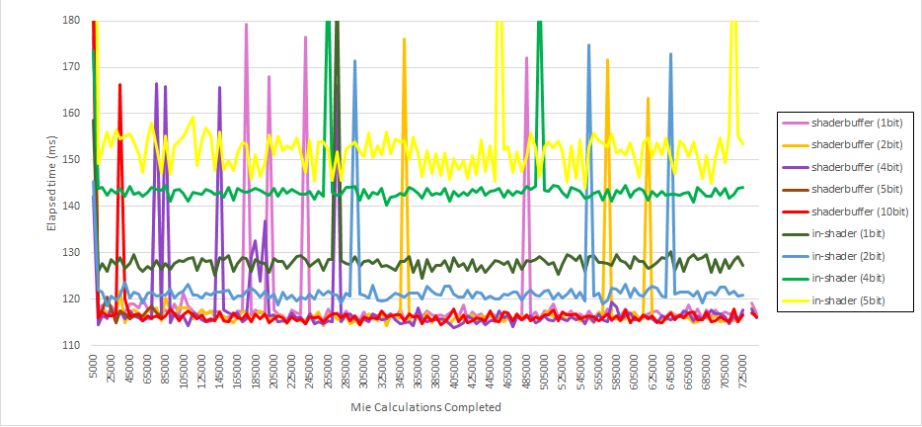
those calculations between shaders. Direct calculation was clearly the slowest of the three methods, with typical times around 600ms. Table-based calculation times were around 115 ms, for a speedup of approximately 6x for the table-based variants as compared to direct calculation. The shaderbuffer implementations were slightly faster than the corresponding in-shader implementations for all table configurations. Note that the results presented here were obtained in an early, un-optimized version of ComputeGlory, so the GPU timing values for the compute shaders should not be compared to timing values elsewhere in this document, only to each other.

Figure 5.7 shows alternative views of the same data. In Figure 5.7(a), the direct-calculation times have been removed, making it more apparent that timing differences between the various table sizes are negligible. Figure 5.7(b) additionally limits the vertical axis to 180 ms for closer inspection. The occasional spikes which exceed 180ms are due to cache misses, where the shader needed to access a different table or section of table than was currently available. These spikes are filtered out in Figure 5.7(c). Table 5.2 shows average filtered times for all methods tested.

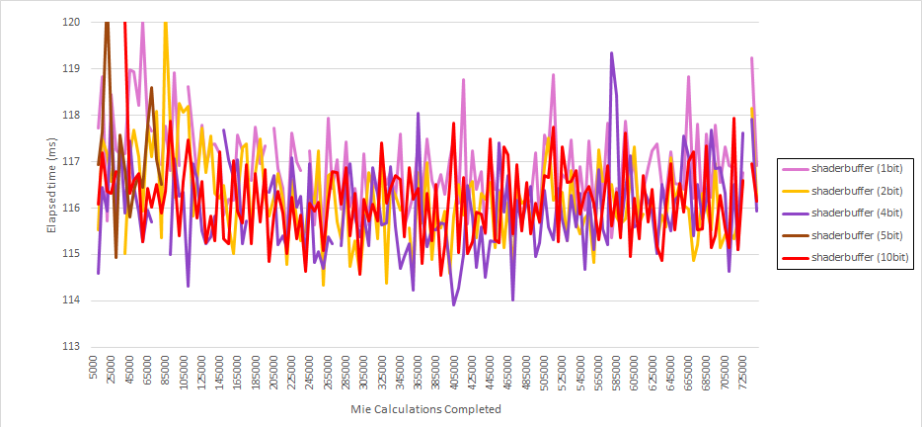
Based on these results, I opted to use 4 tables with 5-bit indices (32 entries) each for subsequent development, as this gave a comfortable tradeoff between table size and the number of tables required.



(a) Full dataset



(b) Y-axis restricted to < 180 ms, hiding startup spikes



(c) Shaderbuffer tables only, data filtered to remove cache-miss spikes

Figure 5.7: Total GPU time for scatter, color-conversion, and merge shaders by shaderbuffer and in-shader Sobol table implementations.

bits	in-shader	shaderbuffer
1	127.8	116.9
2	121.1	116.2
4	143.0	115.9
5	151.7	116.3
10	n/a	116.1
direct	602.4	

Table 5.2: Average times (ms) for scattering, color conversion, and merge shaders. Cache miss spikes have been filtered from this dataset.

Chapter 6

WAVELENGTH / SCATTERING ANGLE SELECTION METHODS

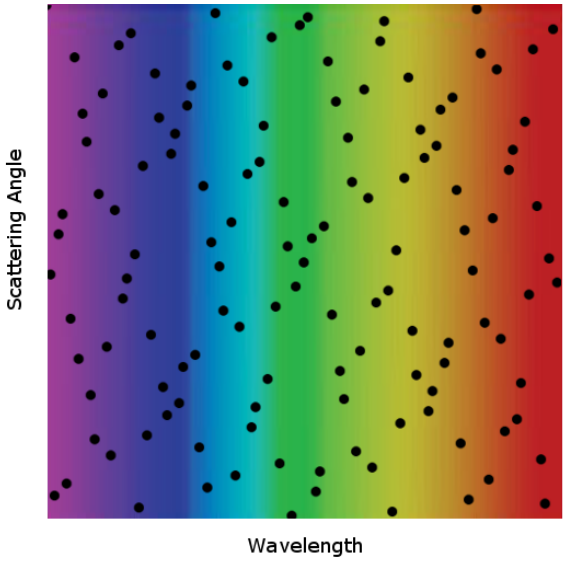
Past work in rendering glories has been done in the field of atmospheric physics, not interactive computer graphics, and has been guided by different goals. For scientific research into the nature of glories, the highest priorities in calculating the colors and intensities of a glory are physical (numerical) accuracy and precision. Speed, although desirable, is a secondary goal. Each pixel in the output image dictates the exact scattering angle to be used in the Mie calculations. For each such angle, a set of wavelengths is generated and the exact output intensity corresponding to each (angle, wavelength) pair is calculated. The wavelengths for Mie scattering are sampled from the visible region of the spectrum, 380 to 730nm. Previous work in atmospheric physics has shown that 500 to 600 wavelengths are necessary for each scattering angle, with the value depending on the parameters of the interacting droplets. The wavelengths are assumed to be evenly spaced across the visible spectrum domain as shown in Figure 6.1(a).

For interactive graphics, however, short rendering times are critical. Also, *perceptual* accuracy is paramount, but exact physical accuracy, although desirable, is less important. A tradeoff of accuracy for speed can be obtained by relaxing the requirement that scattering angles be entirely defined by the pixel grid of the final image. Instead, the entire image can be considered a region of a continuous domain. This region should be sampled thoroughly and efficiently to generate scattering angles, just as the visible spectrum is a region of a continuous domain (the electromagnetic spectrum) sampled to generate wavelengths.

Put another way, the samples for which Mie scattering results are computed should not be thought of as a collection of 1D sets of wavelengths, each set chosen by a 1D sampling technique to cover the visible spectrum for a single predefined pixel, but rather as a single

set of 2D samples chosen to efficiently cover the 2D space defined by the visible spectrum and the applicable range of scattering angles for the entire image, as in Figure 6.1(b).

(a) Evenly spaced 1D samples across wavelength dimension.



(b) Well-distributed 2D samples across wavelength x scattering-angle dimensions.

Figure 6.1: Sampling methods for Mie scattering input values.

The method of selecting the (λ, θ) pairs for Mie calculations is a central decision to the entire glory rendering process. Good coverage of the sample space is essential, but as Mie calculations are time-consuming, it is important to perform no more than necessary to achieve the desired fidelity. The following sections discuss the pixel-oriented method used in prior physics work, which I refer to as "WvPx" (wavelengths per pixel), and the 2D sampling method developed for ComputeGlory, "2DSobol". Finally, I compare the rendering time and error convergence of the two methods.

6.1 1D pixel-oriented "WvPx" method

Each pixel's location in the image determines a scattering angle. For each angle, a set of wavelengths is selected. Each set has the same number of wavelengths as all the others, but not necessarily the same wavelengths. I refer to this method as "1D" even though samples are two-dimensional (λ, θ) pairs, because only the wavelength dimension is selected by sampling. The θ values for the scattering angle dimension are pre-determined by the pixel resolution of the image.

Initially I focused on optimizing the set of wavelengths used in Mie calculation for each pixel. Specifically, I tried to improve the distribution and reduce the number of wavelengths required for each scattering angle.

The simplest way to select the wavelengths is to space them evenly in increasing order across the visible spectrum domain (380 to 730nm) as shown in Figure 6.1(a). This method was used in early implementations such as BHMIE (Bohren and Huffman 1983). With this method, each pixel uses the same set of wavelengths for Mie scattering.

A slightly more complex option is to randomly sample wavelengths using a uniform distribution. There are numerous options for generating the random values required, including pseudorandom and quasirandom sequences, such as a Sobol sequence. The scattering angle, pixel index, or other per-item value can be used to seed or access the random values, producing different wavelengths for each pixel. These are discussed in more detail in Section 2.10.

Figure 6.2 shows the resulting ΔE^* error values from using three different 1D sampling methods at sample counts ranging from 10 to 150 wavelengths per pixel. Although uniform sampling with TEA converges more smoothly than the other two methods, uniform sampling with Sobol gives lower error at very small sample counts, and is the method selected for most of the subsequent tests where the WvPx method is used.

Table 6.1 illustrates a key difference between the equal-spacing method and the other two. For a given wavelength count, all pixels of the equal-spacing method use exactly the

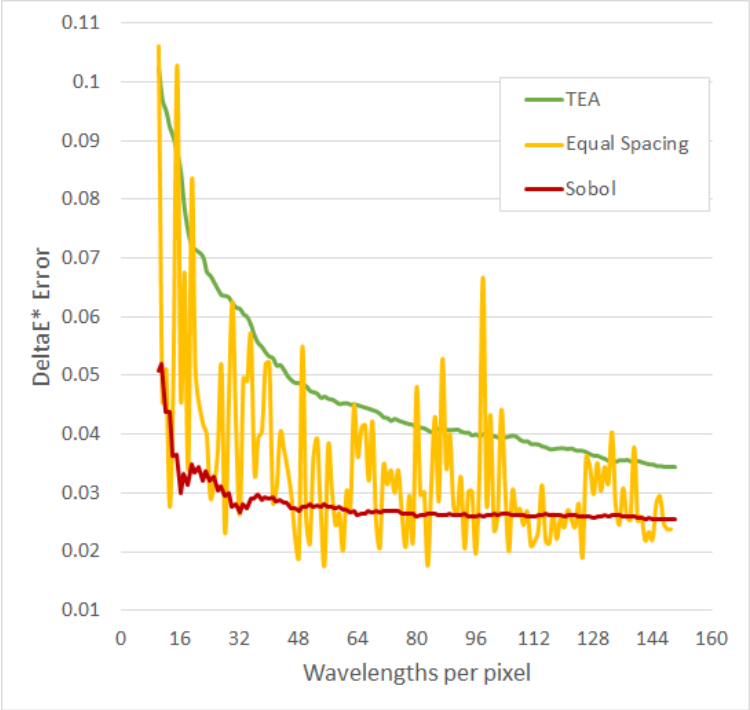


Figure 6.2: Error behavior for three variants of WvPx method, from 10 to 150 wavelengths per pixel.

same set of wavelengths, but changing the wavelength count by only a few wavelengths results in the render using a completely different set of wavelengths. As a result, the two rendered images in the top row of Table 6.1 differ by an overall color shift. For the other two methods, the set of wavelengths computed by each pixel is determined separately, and the result is visual noise. Color discrepancies between pixels of the radial slice manifest as rings in the final image, but the overall color balance of the image does not shift noticeably with small changes in sample count.

There are numerous other selection methods possible and variations of those methods. For example, selecting equally-spaced wavelengths then jittering them on a per-pixel basis produces a slightly different set of wavelengths for each pixel's Mie scattering calculations than for its neighbors. This work is not intended to be an exhaustive investigation of these methods; rather, the three demonstrated here were chosen as representatives of the many options.

	10 wvs/px	14 wvs/px
Equal spacing		
TEA		
1D Sobol		

Table 6.1: Images generated for 1D WvPx sample selection using varying methods and wavelength counts. Northeast quadrants shown.

Importance sampling was introduced in Section 2.9. This technique is widely used in computer graphics as it offers an alternative to uniform spacing or random sampling from a uniform distribution, and can produce good results with far fewer samples than those methods. Instead, a random value is used to sample from a non-uniform probability distribution which is focused (higher probability) in the areas of the domain which are expected to contribute most significantly to the result. Importance sampling can be thought of as sampling from a probability distribution which has been generated from one which is uniform by warping it into one which has higher probabilities in the "important" regions.

The effectiveness of the importance-sampling estimator function depends on how well it represents the actual output. I initially planned to use the human visual perception profile as an estimator function to concentrate sampling effort on the regions of the visible spectrum where human vision has the greatest response. This failed because it is not actually a good

estimator of the output, i.e. the glory color banding pattern. The strongest color output depends on the scattering angle being computed, therefore a two-dimensional estimator is needed, which takes into account both the wavelength and the scattering angle for each sample point. For similar reasons, a three-part estimator based on the human visual system's three color receptors' sensitivity curves (see Section 2.3) was ineffective.

I developed a 2D importance-sampling estimator by fitting a function to the output of a high-resolution, high-sample-count rendering. However, this added quite a bit of complexity to the wavelength selection algorithm and did not provide the desired convergence improvement. This function is discussed more fully in Chapter 7, where it is revisited as the approximation function producing a base image for incremental rendering.

I also developed a 2D estimator based on the same rendered data as the approximation function, but stored as a lookup table of cumulative density function (CDF) values for direct use in importance sampling. As the function generating the data was not amenable to analytical integration, I numerically integrated the probability distribution function, organized so that the 2D importance sampling algorithm could rapidly retrieve the angle and wavelength corresponding to a desired CDF value. This method also offered little if any convergence improvement over sampling from a uniform distribution.

In the end, ComputeGlory does not use importance sampling for Mie scattering sample selection. It uses importance sampling only to select samples for blending when sweeping a radial slice of the glory to form a full circular image.

6.2 2D Sobol sampling method

The previous section describes a two-phase process of selecting samples for Mie calculations: first using the pixel structure of the image to determine the scattering angles, then selecting wavelengths for each angle. The wavelengths were selected as samples drawn from a evenly-distributed one-dimensional sequence, typically generating the sequence index from the scattering angle and within-cell ID, which is an index over the set of wavelengths used

by the current scattering angle. This process was performed independently for each slice pixel.

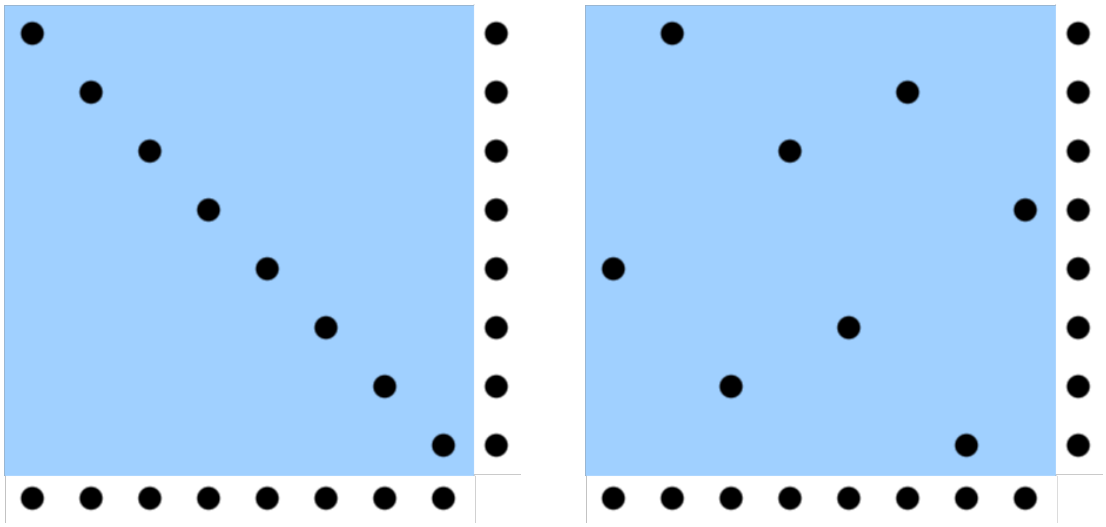
The alternative approach is to view the sample-selection process as a single two-dimensional problem: selecting points from a 2D space whose axes are wavelength and scattering angle.

Variance and discrepancy, and the unwanted artifacts they produce in sampling sequences, were discussed in Chapter 5. In ComputeGlory, the cells into which samples are placed are the slice pixels, and variance within cells appears as clustering or gaps in the collection of wavelengths chosen for each pixel. Between-cell problems are visible when adjacent slice pixels have too-similar or too-different distributions of wavelengths. For example, each pixel may draw a little more from one region of the spectrum and a little less from others, but as long as each pixel concentrates on a different region than its neighbors, the overall result is balanced. However, if several adjacent pixels concentrate on the same wavelength interval, the resulting error is quite noticeable.

Effective coverage of the 2D sample space requires minimizing clumps and gaps in the wavelength sample distribution between adjacent slice pixels as well as within each pixel. This is accomplished by regarding cell index (slice pixel location / scattering angle) not as a parameter in a 1D sampling technique, but as a sample drawn from an additional dimension of the same multi-dimensional low-discrepancy sequence. Then a single Sobol sequence index can be used to sample values from two dimensions of that sequence. For ComputeGlory, it is convenient to use dimensions 2 and 3, although other dimensions could be used.

That this process uses two dimensions of *the same sequence* is important. Using two independent one-dimensional low-discrepancy sequences as the axes of our 2D sampling space would not provide the same guarantee of samples being evenly distributed in 2D, only in their projections onto each axis. Figure 6.3 provides an extreme example to illustrate this important distinction. In Figure 6.3(a), the sample points are uniformly spaced in the

vertical and horizontal directions, so their 1D projections are evenly distributed without clumps or gaps, but viewed as 2D samples they clearly do not cover the sample space in a useful manner. In Figure 6.3(b), samples are evenly distributed along both 1D projections and in the 2D space.



(a) Poor 2D distribution. (b) Well-distributed 2D samples.

Figure 6.3: Both sample distributions give evenly-distributed projections onto each 1D axis, but the 2D distribution on the right covers the sample space much more effectively.

While the smallest cohort of samples the WvPx method processes is one wavelength per pixel (1449 samples, for a 512x512 image with 4x slice resolution), the 2D Sobol sampling process is able to work at single-sample granularity. This is useful in a research context, for example, to determine the exact number of samples required to achieve a target error value by proceeding sample by sample and assessing the error after each one. Of course, in production use the overhead of single-sample cohorts would be completely unacceptable, but cohorts smaller than slice pixel size may be useful. Because the 2D Sobol method does not tie cohort size to the number of pixels in the radial slice, there is no guarantee that the total number of samples processed will be a multiple of the slice pixel count, and therefore the number of wavelengths for each final image pixel may vary. However, choosing an appropriate 2D sampling function ensures that no pixel is greatly under- or over-represented

in the overall collection of samples. Furthermore, no wavelength (or more accurately, no region of the spectrum) is under- or over-represented.

The compute shader functions in `wavelength.glsl` and `sobol.glsl` implement a version of Sobol sampling which produces [0,1) quasirandom value pairs for two correlated dimensions of a Sobol sequence. These pairs are used to select wavelength and angle by uniformly sampling the appropriate intervals. The result is a collection of samples that are evenly distributed in each dimension and in the 2D space.

Chapter 5 describes various techniques for generating and working with Sobol sequences. The 2D Sobol wavelength selection method uses the forward (standard) Sobol implementation to generate (λ, θ) sample pairs for Mie scattering and color conversion. The WvPx sample selection method uses cell-indexed Sobol (see Section 5.3) to generate samples from pixel coordinates and within-pixel index.

ComputeGlory also uses the technique discussed in Section 5.5 to precompute 5-bit partial lookup tables offline for a 32-bit Sobol sequence, thereby reducing runtime sequence access to a small number of table lookups and bitwise operations.

6.2.1 Splatting

The choice of sample selection method affects the parallel structure of not only the scattering phase of ComputeGlory but also the color conversion and accumulation phase. When using the pixel-organized WvPx method, the scattering angle of each Mie calculation corresponds to the center of a slice pixel. The scattering results apply to that pixel alone, and the data produced by all the shader invocations of the scattering phase is grouped by pixel. For the 2D Sobol method, the scattering angle is sampled from a continuous domain. It is highly unlikely that a sample lies exactly at the middle of a pixel, therefore each sample typically straddles multiple pixels. Generally, these are not the same pixels as the adjacent samples in the generated sequence, as the Sobol sequence generates scattering locations in an order that jumps around the domain. The color value of a sample must be applied to each pixel

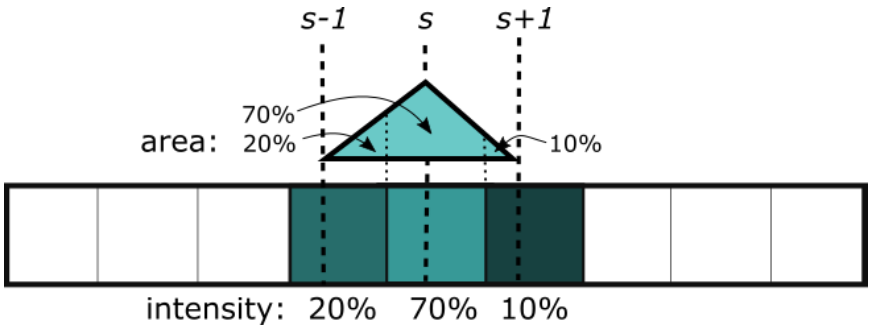


Figure 6.4: Splatting one sample into pixel array using a triangular filter 2px wide. As more samples are splatted, the pixels fill in with multiple wavelengths contributing fractional intensities.

it covers completely or in part, weighted by the shape of a sample filter and the extent to which the sample covers that pixel. This technique of splitting up value based on coverage is known in graphics as *splatting*. It is widely used to render quantities which are represented by a dense distribution of samples across the rendering domain. One can imagine painting a tiled wall by shooting many blobs of paint at it. Each paint blob partially covers multiple tiles in proportions that depend on its contact location and the shape of its splash. In this case, the color value is being splatted into a one-dimensional array of pixels as shown in Figure 6.4. Once color conversion and splatting are complete, the results have the same format for both sample selection methods: a 1D array of slice pixels, each with a single associated color. All downstream shaders are unaware of the sampling method used to produce the data they receive.

`convertColor-splat.glsl` contains the function which performs this splatting. It is called once per pixel of the high-resolution slice, and loops over all samples in the current cohort seeking ones whose regions of influence overlap the indicated pixel. The data and list-traversal cost of this step could potentially be reduced by sorting the sample array between scatter and color/accum phases, so that each shader doesn't have to inspect every sample. However, the cost of the splatting code is so small compared to the scattering calculations that for this work I chose simplicity over a trivial improvement.

The shader retrieves the intensity and wavelength for each sample. It applies them us-

ing a triangular filter region 2 pixels wide, so a sample calculated at location s contributes to all pixels that have full or partial coverage of the interval $[s-1, s+1]$, as shown in Figure 6.4. The area of the fractional overlap of the triangular region with each pixel determines the weighting factor for this sample's contribution to the pixel. In the diagram, 20% of the triangular filter region overlaps the left pixel, so when the shader processes this pixel it accumulates color as if the pixel had received all of a sample with the same wavelength but 20% of the actual intensity value. Similarly, the middle pixel gets 70%, and the right pixel gets 10%. Weights sum to 100% to ensure conservation of sample intensity. The wavelength is transformed into XYZ color space, and the result is scaled by the fractional intensity. As more samples are splatted into the array, each pixel accumulates multiple partial samples, then converts the total XYZ to RGB to yield a final color value.

6.3 Results: Timing, image quality and number of Mie calculations

All tests in this section were performed using a 512x512 image, 4x slice resolution, and standard glory physical parameters as defined in Section 3.2.4. Columns of result data tables that are used as input to later tests are color-coded to help the reader follow the flow of information. No specific meaning is attached to the choice of colors.

6.3.1 Test 1: Timing and error results for same-size cohorts

As an initial investigation, I performed glory renderings using the same numbers of Mie calculations for both methods: 600 wv/px (869,400 total samples), which is above the theoretical minimum number of wavelengths per pixel required, and 300 wv/px, half that amount. The calculations are broken into smaller groups, called cohorts, and distributed across multiple rendering frames, as discussed in Chapter 7. The cohort sizes chosen represent performing all 300 or 600 wavelengths in one frame, in 2 frames, in 4 frames, and so on. In all cases ComputeGlory reported the total elapsed wall-clock time and the total GPU shader time spent in the Mie scattering shader.

wvs/px		WvPx		2DSobol	
total	cohort	wall clock	scatter GPU	wall clock	scatter GPU
600	600	602.034	563.015	19411.6	17231.1
600	300	602.161	563.645	19341.2	17155.6
600	150	615.035	563.272	19365.9	17165.1
600	100	628.436	563.776	19325.7	17118.5
300	300	305.217	282.561	9692.77	8606.34
300	150	318.241	281.558	9654.22	8567.12
300	100	312.818	282.880	9668.55	8566.45

Table 6.2: Timing for both sample selection methods, using equal numbers of Mie calculations split into varying numbers of cohorts with corresponding sizes.

Table 6.2 summarizes the timing results. Clearly, for the same number of Mie calculations, the WvPx method completes much more rapidly than the 2D Sobol method at all sample counts. However, speed is not the only metric of interest. I collected ΔE^* error metrics for the 600 and 300 wvs/pixel cases, and several smaller cases ranging down to only 10 wavelengths per pixel. In each case I used a single cohort comprising the entire collection of Mie calculations. Table 6.3 and Figure 6.5 present the error results. At all sample counts examined, the 2D Sobol method produces much lower-error images from the same number of calculations, 14% to 26% lower than the corresponding WvPx image. Table 6.4 illustrates the difference between images produced with 10, 25, and 50 wavelengths per pixel or equivalent 2D sample counts. Error images have been brightened and contrast-enhanced for visibility. Although the error images have similar overall brightness, corresponding to average error, the Sobol error images are less grainy and have fewer strong bright bands (high error regions).

sample count (wvs/px)	WvPx error	2DSobol error	ratio (2DS/WvPx)
600	0.02375	0.02043	0.86
300	0.02467	0.02056	0.83
100	0.02622	0.02163	0.82
50	0.02759	0.02372	0.86
25	0.03282	0.02709	0.83
10	0.05083	0.03768	0.74

Table 6.3: ΔE^* error compared to reference solution for both sample selection methods.

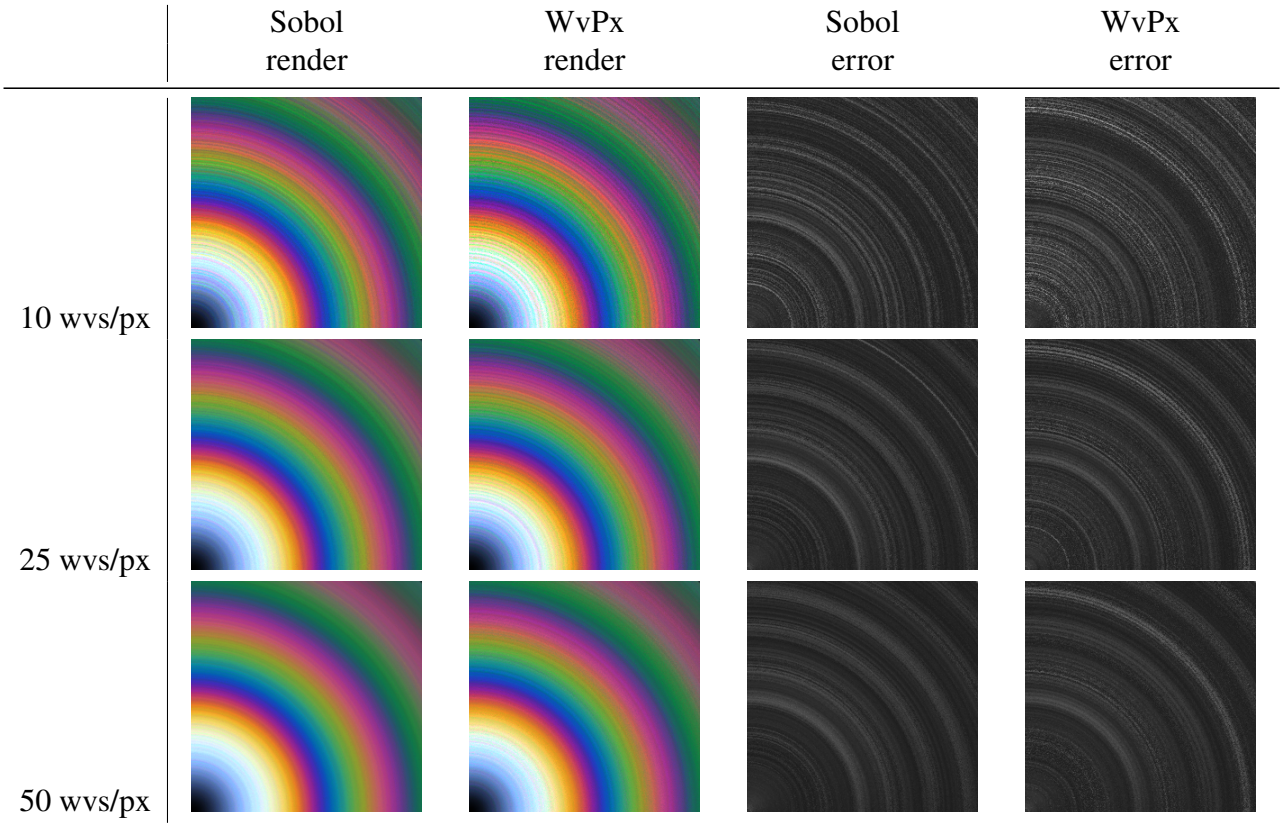


Table 6.4: Rendered images and ΔE^* error images produced by both methods using very low sample counts. Northeast quadrants only are shown.

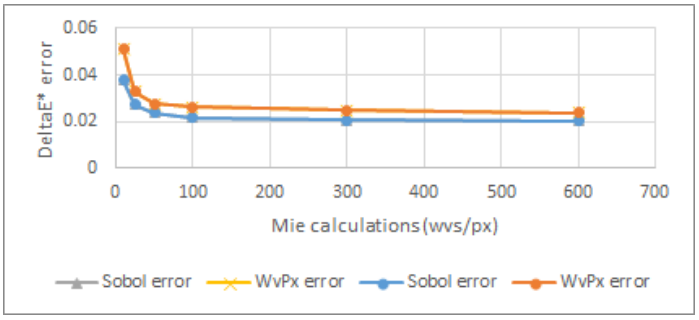


Figure 6.5: Error convergence behavior for WvPx and 2DSobol.

6.3.2 Test 2: Timing results for same-size small cohorts

Next, I examined the time consumed by each method when restricted to a very small number of Mie calculations. Figure 6.6 and Table 6.5 present timing results for both methods using the same number of total Mie calculations, ranging from 10 wavelengths per pixel (14490 samples) to 100 wv/px. (144,900 samples). For this and subsequent tests, I recorded only the GPU time consumed by the scatter phase, not the wall-clock elapsed time. The variations in the latter due to unrelated system activity tended to overwhelm the small computation times of these low-sample-count tests.

Again, for the same number of Mie calculations, the WvPx method completed much more rapidly than the 2D Sobol method at all sample counts. These results were recorded as a baseline for comparison to improved results in Test 4.

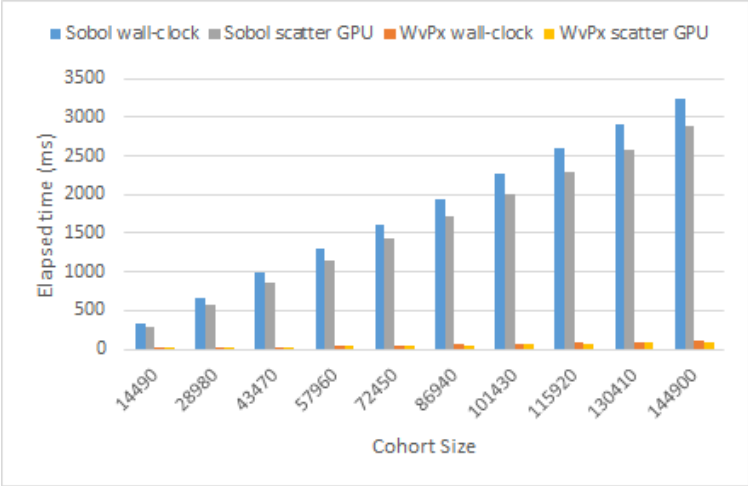


Figure 6.6: Comparison of total (wall-clock) and scattering-shader time for WvPx and 2D Sobol wavelength selection methods.

Mie calculations (wvs/px)	(samples)	WvPx time	2DSobol time	ratio
10	14490	9.54	287.24	30.11
20	28980	18.86	574.46	30.45
30	43470	28.18	861.01	30.55
40	57960	37.54	1147.63	30.57
50	72450	48.42	1434.79	29.63
60	86940	56.31	1721.43	30.57
70	101430	65.57	2008.59	30.63
80	115920	75.06	2295.19	30.58
90	130410	84.76	2582.05	30.46
100	144900	93.80	2875.36	30.65

Table 6.5: Timing results in milliseconds for GPU scatter shader using both sample selection methods.

6.3.3 Test 3: Cohort sizes for same error results

For a comparison of the two sample selection methods which fairly compared sample counts that produce equivalent images (images which are not identical, but give the same error measure), I first determined the number of Sobol samples needed to achieve same error as using WvPx with each of the Mie calculation counts from the previous test (10wv/px, 20wv/px, etc.) This was accomplished by setting the end criterion to be the render error,

with a target of the corresponding WvPx error values, and recording the number of Mie scattering calculations needed by the 2DSobol method to reach that error value. These results are shown in Figures 6.7 and 6.8 and Table 6.6.

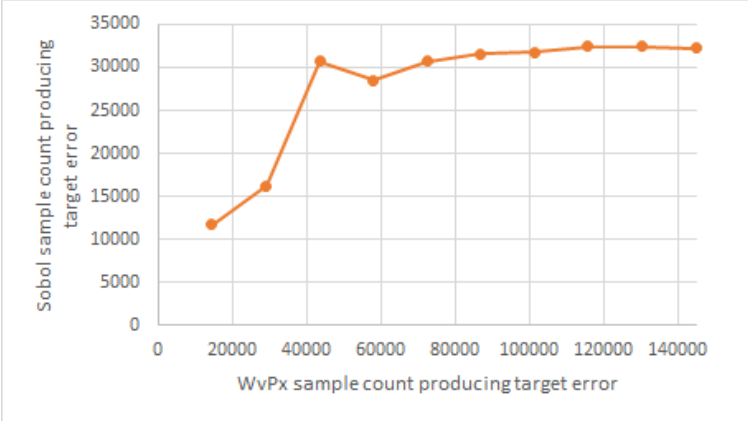


Figure 6.7: Sample counts for 2DSobol to match error of given WvPx counts.

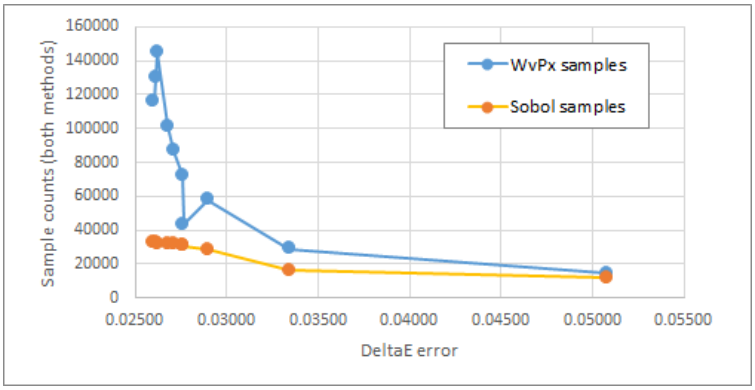


Figure 6.8: Sample counts for 2DSobol to match error of given WvPx counts, plotted against the error values.

In Figure 6.7, the third data point is unusually high compared to its neighboring data points. The corresponding row of Table 6.6 shows that this is because the error target for this test run, 0.02762, was unusually low, requiring more Mie calculations to reach that error target. This error value was the outcome of running the WvPx method for 30 wv/px.

A closer look at the corresponding curve in Figure 6.2 provides the explanation. This outlier value is caused by aliasing due to data collection at 10 wv/px intervals. Looking at Figure 6.2, which was generated from data collected at 1 wv/px intervals, it is apparent

WvPx wvs/px	WvPx samples	ΔE^* error	2DSobol samples	samples ratio
10	14490	0.05083	11700	0.81
20	28980	0.03345	16200	0.56
30	43470	0.02762	30700	0.71
40	57960	0.02897	28500	0.49
50	72450	0.02759	30700	0.42
60	86940	0.02712	31600	0.36
70	101430	0.02680	31800	0.31
80	115920	0.02600	32400	0.28
90	130410	0.02615	32400	0.25
100	144900	0.02622	32200	0.22

Table 6.6: Sample counts required for 2DSobol method to match error of given WvPx counts.

that the error vs. sample count graph for WvPx using a 1D Sobol sequence to select the wavelengths is not smooth. It exhibits oscillations at two scales: small ones on a 2-wv/px cycle and larger ones on a 16-wv/px cycle. These oscillations demonstrate a known characteristic of the Sobol series: when enough Sobol samples are generated to fill a grid of a given grid size, they are evenly distributed, but intermediate partially-filled grids may not be. For ComputeGlory, this means that half the cells of the wavelength/angle 2D space are filled first, then the other half, at one or more cell resolutions which depend on the particular Sobol dimensions and direction numbers used. The observed result is that as more and more samples are processed, the rendered image accumulates error over the first half of each cycle, then resolves it over the second half. The error value reported in Table 6.6 is in fact consistent with the more detailed observed behavior.

I did not consider it necessary to repeat this test with smaller sample-count increments as the goal was simply to identify the overall trends. Once the apparent anomaly was confirmed as caused by Sobol sequence fill order rather than data collection error, the 10 wv/px increments used for this test were sufficient. A potential remedy for the Sobol sequence fill order problem is discussed in the future work section of Chapter 8.

ΔE^* error	WvPx samples	WvPx scatter	Sobol samples	Sobol scatter	samples ratio	time ratio
0.05083	14490	9.33	11700	231.84	0.8075	24.30
0.03345	28980	18.42	16200	320.94	0.5590	17.01
0.02762	43470	43.90	30700	607.97	0.7062	21.58
0.02897	57960	58.41	28500	564.43	0.4917	15.03
0.02759	72450	60.74	30700	607.97	0.4237	12.56
0.02712	86940	74.00	31600	625.88	0.3635	11.12
0.02680	101430	78.70	31800	629.73	0.3135	9.60
0.02600	115920	80.36	32400	641.64	0.2795	8.55
0.02615	130410	83.17	32400	641.64	0.2484	7.57
0.02622	144900	91.76	32200	637.67	0.2222	6.80

Table 6.7: Timing results for both methods, using sample counts which were determined to give comparable error results. “Scatter” columns are GPU time in milliseconds for scatter shader. Ratios give 2DSobol value divided by corresponding WvPx value.

6.3.4 Test 4: Timing results for same-error sample counts

Test 3 determined equivalent sample counts for the WvPx and 2DSobol methods. I ran a new render using the 2DSobol method with each of those sample counts, and recorded the wall-clock and compute shader times. These results are shown in Table 6.7 and Figure 6.9. The right-most column of Table 6.7, showing the ratio of scatter shader time between the WvPx and 2DSobol methods, should be compared to the same column of Table 6.5 from Test 2. In that test, where cohorts of the same size were being compared, the time ratios were on the close order of 30x (29.63 to 30.65). Here, using cohorts which produce images of the same error level, the time ratios range between 6 and 24. WvPx is still faster for the same error level, but not by such a large factor.

It was necessary to re-render the images to obtain the timing data. Recording timing from the render in which the number of samples was determined was not appropriate because that render was deliberately slow. To determine the sample count closely, I set ComputeGlory to run an incremental render with a very small cohort size and compute the error measure after each cohort. This is quite a tedious process and not representative of typical usage or performance. Instead, when the desired number of Mie calculations is

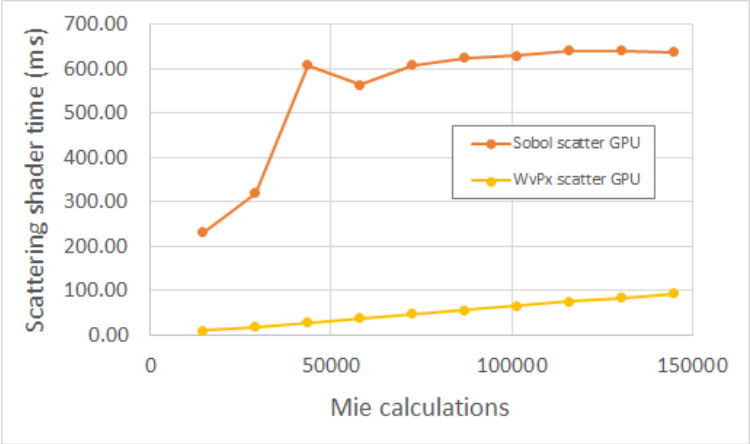


Figure 6.9: Timing results for both methods, using sample counts which were determined to give comparable error results.

known, a render can be run with the end criterion set to the number of calculations. The images can be produced by a small number of large cohorts, with no need to compute the error measure after each incremental step.

The resulting Mie calculations produce visually and analytically comparable results using far fewer samples than the pixel-driven method, between 19% and 78% fewer for the cases examined. The error levels previously achieved with 14490 to 144900 samples (10 to 100 wavelengths per pixel) can now be achieved with the reduced sample counts shown in Table 6.6. However, even though the 2D Sobol method performs far fewer Mie calculations, the cost of computing the Sobol sequence values causes this method to run several times slower to produce equivalent images.

Looking at it another way, for fixed small sample counts, the 2D Sobol sample selection method does more with less, and produces far lower error results than the 1D WvPx method, but at the cost of longer running times.

6.3.5 Test 5: Sample counts for 2D Sobol method with low error targets

The samples ratio between WvPx and 2D Sobol (second-from-right column of Table 6.7) decreased dramatically as the error target became more challenging. As a follow-up test,

I explored the sample counts needed for the 2D Sobol method to reach even lower error targets, in order to characterize the sample-count behavior of this method. I looked at a range of target error values substantially below the smallest error target used in Sections 6.3.3 and 6.3.4, which was 0.02600. The intended target values ranged from 0.024 down to 0.020, and results are shown in Table 6.8. However, no results are available for target error values below 0.0205 because these tests failed to complete after running for approximately 25 minutes. Figure 6.10 shows the rapid increase in sample count requirements as the target error approached 0.0205.

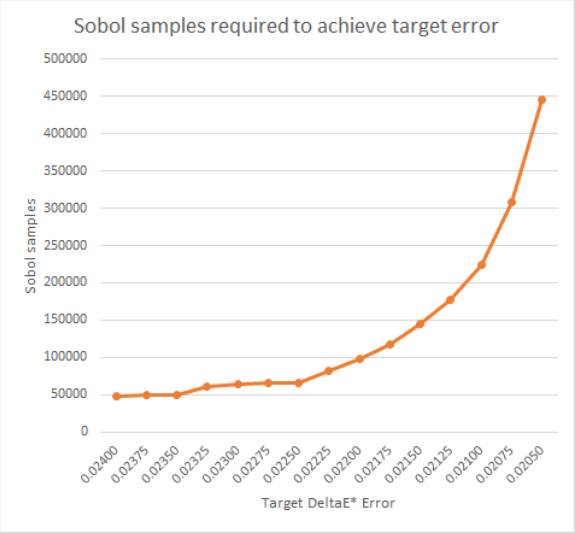


Figure 6.10: Sample counts for 2DSobol to produce renders with specified ΔE^* error.

Future investigation into the 2D Sobol method should include a study of the effects on this convergence behavior if image parameters (size, slice resolution) and physical parameters (droplet size, index of refraction, input spectrum, field of view) are changed.

target error	Sobol samples
0.02400	48500
0.02375	48900
0.02350	49100
0.02325	60800
0.02300	63600
0.02275	65100
0.02250	65500
0.02225	82400
0.02200	98300
0.02175	116700
0.02150	145300
0.02125	178000
0.02100	224400
0.02075	308200
0.02050	445500

Table 6.8: Sample counts for 2DSobol to produce renders with specified ΔE^* error.

Chapter 7

INCREMENTAL RENDERING

7.1 Merits of incremental rendering

Previous chapters have introduced several methods of reducing the number of Mie calculations required to produce a perceptually-acceptable glory rendering. These methods all reduce the number of Mie calculations performed per frame by reducing the total number of calculations required. Another way to reduce the number of Mie calculations performed per frame is to split them up and execute only a subset (a "cohort") of the calculations per frame, achieving the same end result after a small number of frames.

Incremental rendering is not merely a coping strategy to be applied when the complete Mie scattering calculation takes too long to finish in a single frame. It has several characteristics which make it desirable on its own merits.

Because only a portion of the total samples are computed in each frame, incremental rendering reduces the amount of data per frame shuttled between CPU and GPU. As this transfer is one of the bottlenecks in any application using the GPU, keeping it small is preferred. It also reduces the amount of data needed by some compute shaders, thus reducing the size of each invocation and potentially increasing the parallelism achieved.

Breaking a large set of calculations into smaller cohorts affords the opportunity to assess completion or convergence after each cohort, and stop when a specified criterion has been achieved, which may be sooner than if the full calculation were run to completion. One of the completion criteria used in ComputeGlory is the ΔE^* difference between the current result and a previously-computed reference solution. In other applications a reference solution may not be available, so other quality criteria could be applied, such as difference from the most-recent previous rendering result, or within-image variance measures. It is

also possible to stop calculations after a given amount of GPU time has been consumed, or after a given interval of wall-clock time has elapsed.

An incremental render can respond to time-varying input which comes from scene content, computing each cohort with different physical parameters to account for situations such as the sun setting (which affects the solar intensity spectrum used as input to the glory rendering pipeline) or the fluid droplet size changing as the underlying cloud ages.

Finally, incremental rendering supports adaptive cohort sizing, changing the number of Mie calculations performed each frame. Cohort size flexibility lets the system maximize the calculations completed within its allotted time budget. The cohort can also be resized in response to time-varying system load, ensuring that the glory rendering shaders remain within their allotted time budget when that budget changes, as may occur within interactive applications when other objects enter or leave the scene.

7.1.1 Temporal antialiasing

Temporal antialiasing can be regarded as amortizing the cost of supersampling by sharing results over multiple frames, just as spatial antialiasing can be regarded as sharing results across multiple pixels. In the case of glory rendering, the amortization across frames includes high-frequency sampling in both the wavelength and scattering-angle dimensions; that is, the visual spectrum and scattering-angle spaces are being sampled very finely, but spreading the calculations and their results across frames reduces the amount of calculation to be performed in a single frame.

To break up glory rendering across frames, there are several considerations: how many samples to handle in each incremental cohort, which samples to include in a given cohort, and how to incorporate the results of each cohort into the existing results.

Each of these will be discussed in subsequent sections.

7.1.2 Cohort size

The ComputeGlory application exposes a user parameter which controls the cohort size by specifying how many Mie scattering samples will be calculated per incremental step. This value may be input as either a count of 2D Sobol samples or as wavelengths per pixel, depending on the sample selection method being used. Section 7.3 introduces adaptive cohort sizing, a technique in which the size of the current cohort is adjusted frame by frame based on recent system performance and current time budget. When adaptive cohort sizing is active, the *cohort size* user parameter is interpreted as the initial cohort size. The minimum and maximum allowed cohort size can also be specified.

7.1.3 Cohort selection

The algorithm for selecting the samples to process in each cohort is numerically unchanged from the single-frame case. The ComputeGlory host application keeps track of the sequence index of the last sample processed, and provides the shaders with the start index of the new cohort. Combining the start index with their instance IDs, which only tell them which sample out of the current cohort, the shaders can determine which samples they're working on in the global sequence. The global index is needed in order to determine what wavelength (and, for 2D Sobol, what angle) to process.

7.1.4 Accumulating results

During each frame of an incremental render, the scattering and color conversion results for a new cohort of samples are computed. The result is a buffer containing a color value for each pixel of the radial slice. The per-pixel color values are merged into the previous results by the `accumulateColor.glsl` compute shader, which requires a parameter indicating the blend weight for the new buffer. The blend weight is determined by considering the number of Mie scattering calculations in the current cohort as a fraction of the total number of Mie calculations performed up to this point (including the current cohort). Blend weight auto-

matically responds to the frame-by-frame differences in cohort size which may be imposed by adaptive cohort sizing.

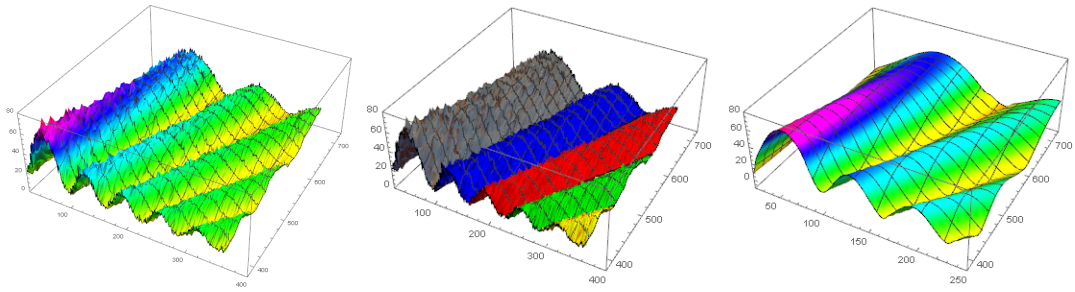
7.1.5 Timing and output issues

When a single-frame render is broken into multiple sub-frames, total elapsed time is expected to be longer, since the core calculations remain the same but supporting CPU activity and some GPU activity (e.g. render-to-screen passes) are performed more than once. The expected per-frame GPU time also increases slightly, tallying more than simply $1/n$ of the single-frame GPU time, because of two types of increased overhead. First, data and control must be passed back and forth n times; second, additional GPU activity is required to merge each successive cohort result into the overall result. Section 7.4 looks at the GPU processing cost for ComputeGlory in more detail.

Recent work in real-time volumetric cloud simulation illustrates this technique. In Guerilla Games's Decima game engine (Schneider 2015), the NUBIS cloud system provides an example. NUBIS partitions its calculations into a large grid, which is broken into 4x4 subgrids. Only one cell of each 4x4 subgrid is updated each frame. The frame result is merged into the ongoing results, which are first reprojected to account for camera motion between frames. Because of the overhead of partitioning, reprojecting, and merging, the theoretical 16x speedup yields about 10x in practice, with negligible visual difference compared to rendering the entire 4x4 subgrid on each frame.

7.2 Approximation for first frame

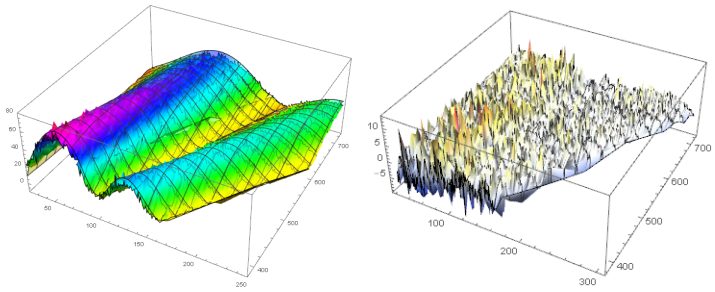
The simplest variant of incremental rendering begins with an empty (black) image. It computes the first cohort's result and displays that for the first frame. However, especially with WvPx sample selection, which is not guaranteed to be evenly distributed for small sample counts, the first cohort may contain so few samples that its output is visibly incorrect and unacceptable.



(a) Mie scattering data

(b) Dataset segmented along troughs.

(c) Approximation function



(d) Mie data and approximation function, peaks 0 & 1

(e) Error between data and approximation, peaks 0 & 1

Figure 7.1: 2D function fit to Mie scattering data from high-resolution, high-wavelength-count reference solution.

7.2.1 2D approximation function

A more satisfactory alternative to a black first frame is to begin with an approximate result which can be computed in a single frame, then refine this approximation over the next few frames with successive cohorts of actual Mie scattering results. The `scatter-approx.glsl` shader computes a function previously fitted to data from the reference solution, which is a high-resolution, high-wavelength-count glory rendering. This dataset is shown in Figure 7.1(a). The resulting approximation function, shown in Figure 7.1(c), was developed using Mathematica. To achieve a visually-acceptable fit it was necessary to slice the data into several sections following the troughs of the data, as shown in Figure 7.1(b), and fit the parameters of the function to each peak separately. The first peak (gray) was more difficult to fit, and in a second fitting pass, it was split along the crest into upper and lower slopes to

section	a	b	c	d	e	f
0 (lower)	19.21	-1.34368	1.90862	-9.00753	-0.307148	35.2949
0 (upper)	40.8859	-5.88436	0.701986	-47.3567	-0.105205	26.4019
1	-32.4674	0.285757	-0.26865	-62.4836	-0.0423437	17.1591
2	32.731	-2.61779	0.24609	-84.7096	-0.0281039	17.063
3	-31.8108	1.13739	0.181815	-114.713	-0.0521497	44.494
4	31.1967	-15.3794	-0.201447	-216.599	-0.0599048	47.6297

Table 7.1: Approximation function parameters for each segment of the function. These values were fit to each section of the high-resolution reference data.

Listing 7.1: Shader code to compute approximate glory function. x is the adjusted pixel position, y is the wavelength. The remaining parameters are fitted to each peak of the function.

```
float fit_func(float x, float y, float a, float b, float c,
               float d, float e, float f)
{
  // axis of sinusoid
  float z = a*x/y + b;
  float sinusoid = sin( z ) + 1;

  // sinusoid amplitude envelope: falls off as z grows
  float amplitude = c*z;

  // angle the crests/troughs of the sinusoid, as seen from above
  float wave_tilt = y / (x + d);

  // alters the centerline of the sinusoid, as seen from front
  float lift_tilt = e*x + f;

  return  ( sinusoid * amplitude * wave_tilt + lift_tilt );
}
```

be solved separately. Table 7.1 shows the parameter set for each section of the final function. The approximation shader calls the function shown in Listing 7.1 for each peak and blends smoothly between the individual peak functions so there are no discontinuities along the troughs in the shader output. Figure 7.1(d) shows the shader output overlaid on the source data, and Figure 7.1(e) shows the ΔE^* error between the source data and the approximation function.

The relationship between the physical scattering angle for a particular Mie calculation and the corresponding pixel location in the radial slice is determined by three angular cor-

Listing 7.2: Shader code adjusts input scattering angle to accommodate different angular correspondence parameters than the original dataset.

```
float px2fitpx(float px)
{
  float fitpx = px;

  // slice multiplier: fit function is derived with smult = 2
  fitpx = fitpx * SliceMultiplierEstimator / SliceMultiplier;

  // image size: fit function is derived with ImageSize = 256
  fitpx = fitpx * ImageSizeEstimator / ImageSize;

  // larger view distance == narrower bands
  // default view distance 12.706203f is derived from
  // default scatterSpread of 4.5 degrees)
  fitpx = fitpx * ViewDistanceEstimator / ViewDistance;
  //fitpx = fitpx * ( sin(4.5) / sin(MaxScatterAngle) );

  return fitpx;
}
```

respondence parameters: the output image size, the slice multiplier, and a third parameter which sets the maximum scattering angle depicted by the final image. The third parameter can be thought of as angular field-of-view, and it implicitly defines a view distance for the rendering from the cloud face producing the glory phenomenon. The data used to fit this function was produced by a rendering whose parameters were set to middle-of-range representative values. However, if any of those parameters are set differently for a given rendering, the shader cannot directly use the pixel index derived from its instance id as input for the approximation function. Instead, the pixel index must first be scaled to compensate for the differences in the angular correspondence parameters. The `px2fitpx()` function in the `wavelength.glsl` shader performs the necessary scaling, as shown in Listing 7.2.

The approximation function also requires that its input be adjusted to accommodate glory rendering using a different droplet size than the size used when generating the function-fitting dataset. This adjustment is not needed by the true Mie scattering shaders, as they take droplet size as an active parameter in their calculations, so it is not part of `px2fitpx()`. Instead the `approx_mie()` function in `scatter-approx.glsl` first calls `px2fitpx()`, then applies the droplet-size adjustment to the result, as shown in Listing 7.3.

Listing 7.3: Scatter-approx shader adjusts input scattering angle to accommodate droplet size difference from original dataset.

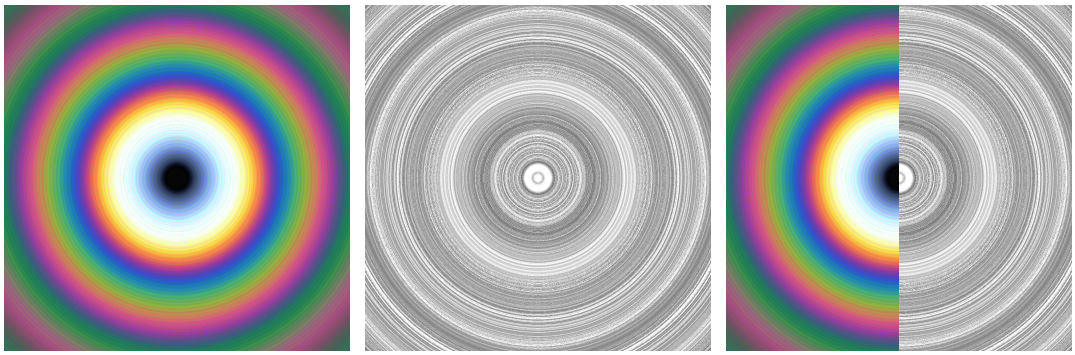
```
// Adjust position (angle) to match the parameters that
// were used to generate the function-fitting dataset
float xx = px2fitpx(float(pidx));

// SCATTERING ADJUSTMENT
// smaller droplets == wider bands
// function was fit with 10micron droplets
xx = xx * float(DropletRadius) / 10.0f;
```

7.2.2 Ghost cohort equivalence

The 2D approximation function in Listing 7.1 was fitted to data from a high-resolution, high-calculation-count glory rendering. The function takes wavelength and scattering angle as input and produces an intensity value, just as the actual Mie equations do. Although using this fit function to derive a probability estimator for importance sampling did not prove effective, it is nevertheless a comparatively cheap estimator for the true Mie calculations.

I extended the incremental sample-cohort model of Section 7.1.2 to include an optional initial estimate generated by computing the approximation function over a small cohort of wavelengths and scattering angles. This "ghost cohort" provides the incremental calculations with a better starting point than an empty buffer. To quantify the improvement, I investigated the convergence behavior of the rendering with and without the approximation base image.

First, I ran the process using the approximation function with only 30 wavelengths per pixel. This partial render yields an image which is recognizably a glory, but contains a good deal of noise. The resulting image is shown in Figure 7.2(a). Its ΔE^* error (compared to the reference solution) is shown in Figure 7.2(b) and has an average value of 0.075. The error image has been brightened and contrast-enhanced for visibility. Figure 7.2(b) combines the previous two images for ease of comparison. The red and blue regions of the rendered image correspond to brighter error bands than the yellow and green regions, indicating that the approximation function demonstrates greater error in the former regions.



(a) Rendered image (b) ΔE^* error (c) Overlay for comparison

Figure 7.2: Approximation function, run for 30 wavelengths/pixel.

Next, I determined the size of the corresponding ghost cohorts; that is, how many Mie calculations were required to produce an image with the same level of error, using each sampling method. I ran incremental renderings using the upward- recurrence Mie shader with the approximation image's error of 0.075 as the end criterion. I used the smallest practical cohort step sizes: 1 wavelength/pixel for the WvPx sampling method and 100 samples for the 2D Sobol method. The WvPx process produced images with equal or lower error after 15 wavelengths/pixel. At 1024x1024 image resolution and 2x slice multiplier, the radial slice contains 1449 pixels, so 15 wavelengths/pixel constitutes $15 * 1449 = 21735$ Mie calculations. The 2D Sobol rendering converged to the desired error level after 14200 samples, approximately 1/3 fewer Mie calculations. 14200 samples correspond to just under 10 wavelengths/pixel, so the reduction is slightly more than the number of Mie calculations required to render 5 wavelengths/pixel.

To examine the behavior of the rendering error with slightly smaller cohorts, I also ran partial renders with end criteria set to slightly fewer wavelengths/pixel and Sobol samples, as shown in Table 7.2. The smaller Sobol sample counts correspond to the number of Mie calculations making up 9 and 8 wavelengths/pixel. As expected, error increased with reduced sample counts.

This tells us that using an approximation image as the base for incremental calculation produces an image of equivalent quality to the image obtained by computing an initial

Sampling method	wvs/pixel or samples	ΔE^* error
WvPx	15	0.067
	14	0.097
	13	0.109
Sobol	14200 (< 10 wvs/px)	0.075
	13041 (9 wvs/px)	0.087
	11592 (8 wvs/px)	0.113

Table 7.2: Ghost cohort error compared to reference solution. The first line of each section is the actual ghost cohort size.

"ghost cohort" of 15 wavelengths/pixel in WvPx mode or 14200 2D Sobol samples. In general, using an approximation as the base for incremental calculation carries the risk that the approximation may bias the final output. I did not pursue this question for ComputeGlory, because the ΔE^* difference between the approximation-based final renders and the reference solution was small enough that the nature of the difference was not of great concern. In contexts where the remaining error is significant, more attention should be given to characterizing the error and identifying bias.

method	base	error target	cohort size	samples req'd	final error
Sobol	none	0.04	100	14100	0.03891
	approx	0.04	100	10200	0.03959
Sobol	none	0.025	100	45000	0.02486
	approx	0.025	100	30700	0.02482
WvPx	none	0.04	1449	21735	0.03567
	approx	0.04	1449	20286	0.03617
WvPx	none	0.025	1449	46368	0.02455
	approx	0.025	1449	44919	0.02439

Table 7.3: Effect of "ghost cohort" approximation on number of Mie calculations required to converge to a specified error level. The WvPx tests used a cohort of 1 wavelength per pixel, or 1449 samples at 512x512 and 4x slice resolution.

7.2.3 Ghost cohort savings

To examine the savings achieved by using an approximation function as the base image for incremental rendering, I selected a much lower error value as end criterion, and compared the number of Mie calculations required to reach that error level when starting with the approximation base versus starting with an empty buffer.

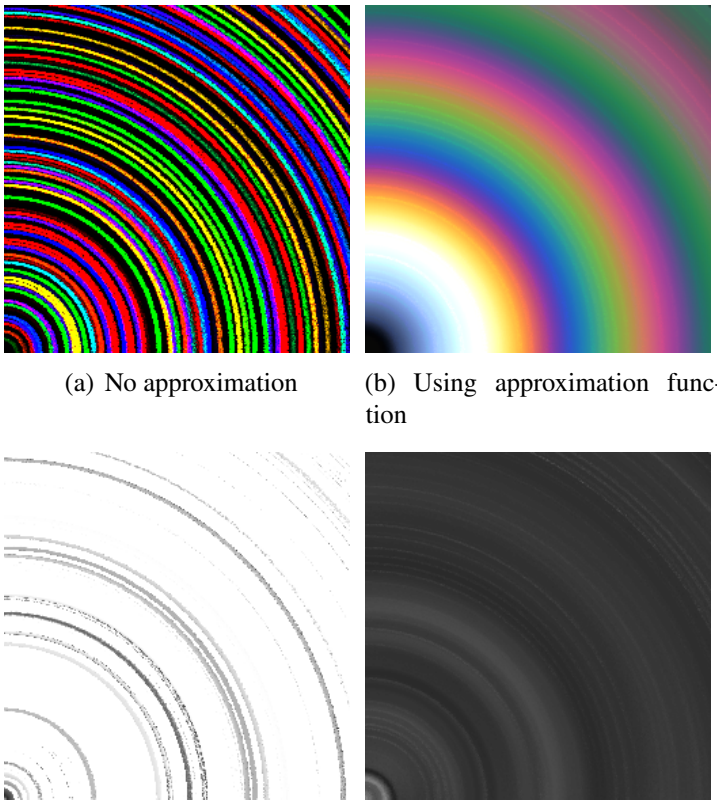
Table 7.3 shows the results of these tests using two different target error values, 0.025 and 0.04, or 1/3 and roughly 1/2 of the error value used in Section 7.2.2 to compare ghost cohort sizes. In all test cases, I used a 512x512 image and 4x slice resolution.

For WvPx, little benefit is seen from using the approximation function. Starting with an empty buffer required 21735 total Mie calculations to reach the 0.04 error level. Starting with the approximation base required only 20286 Mie calculations, a savings of 1449 total calculations, or one cohort, 6.7%. With the 0.025 error target, 46368 calculations were reduced to 44919, again a savings of only one cohort, 3.1%.

For 2D Sobol sampling, greater benefit is observed. Starting with an empty buffer required 14100 Mie calculations to reach the 0.04 error level. Starting with the approximation base required only 10200 Mie calculations to reach the target error level, a reduction of 1900 total calculations or 13.5%. With the smaller error target of 0.025, 45000 calculations

were reduced to 30700, a difference of 14300 calculations or 31.8%.

Figure 7.3 illustrates the effect of using an approximation image rather than an empty buffer for the first frame of Mie scattering results to blend with. The top row shows the rendered output of the first Mie-scattering frame of an incremental render without (left) and with (right) an approximation frame. The bottom row shows the corresponding ΔE^* error images. The error images have been brightened and contrast-enhanced for visibility.



(a) No approximation

(b) Using approximation function

(c) No approximation

(d) Using approximation function

Figure 7.3: Effect of using an approximation function for the first frame of an incremental render sequence. (Northeast quadrants)

7.3 Adaptive cohort size

In many interactive applications, such as games or virtual reality, the total graphics system workload may vary from frame to frame in response to planned application activity and unpredictable user input. As a result, the resources available for glory rendering may vary, and the allowed time budget may be changed by the system of which the glory renderer is a component. Setting a fixed Mie calculation cohort size is unlikely to make the best use of the GPU resources, as it will generally be either too small, in which case better results could have been achieved in fewer frames, or too large, in which case the glory rendering component fails to honor its processing time budget. Adaptive cohort sizing addresses this issue by reassessing the glory rendering performance each frame and adjusting the cohort size to take into account both the recent performance and the next frame's time budget.

Adaptive cohort sizing can be considered a simple form of adaptive task scheduling, in which the individual Mie calculation tasks are scheduled indirectly as a result of their assignment to dynamically-sized cohorts. Assessing the glory rendering system's rendering rate over the most recent few frames enables the cohort-resize code to predict future rendering rate and predict the optimal cohort size to fully utilize the available resources without exceeding the allowed rendering time.

I implemented adaptive cohort sizing only for the 2D Sobol sample selection method. For WvPx, either cohort size changes would be limited to quanta equal to one wavelength-per-pixel, which is too coarse when using large images and high slice multipliers, or large modifications to ComputeGlory's parallel structure would be necessary to enable computing part of a wavelength group in one frame and the rest in the next. As the 2D Sobol method supports cohort size management with single-sample increments, I chose to focus on the Sobol/adaptive case for further study.

7.3.1 Determining cohort size for adaptive cohort sizing

The color accumulation phase of the compute shader pipeline requires a minor modification as it must take into account the potentially-changing cohort size when calculating the fraction to use when blending the new cohort's results into the previous results.

My implementation uses a minimum and maximum cohort size, hardcoded in the host application based on empirical observations during development. In practice, the appropriate values should be determined based on time budget, system load, and other context-specific parameters. As each frame of the incremental rendering proceeds, the system records the GPU shader time consumed by the Mie scattering phase. At the start of the next frame, the average scattering performance rate over the last three frames is used to determine the appropriate size for the next cohort, taking into account the current frame's time budget. If the indicated cohort size is outside the allowed cohort size range, the size value is clamped. Optionally, the system can also restrict the amount by which the cohort size is allowed to change each frame. This cap can prevent large changes in response to momentary performance spikes, which then disrupt performance on subsequent frames and must be corrected. However, it also limits the ability of the cohort size to respond correctly to large changes in the time budget.

I chose to use window sizes of three and five frames for assessing past performance, based on observed behavior of the system on which I was testing ComputeGlory. Likewise, the cohort size ranges and initial cohort sizes I set for my tests were based on previously observed performance. I did not use change- restriction for the data reported below, in order to see how closely the cohort size prediction mechanism could track a changing time budget if given free rein. These parameters and toggles are all user-controllable. They should be set based on observation or analysis of system performance, as well as policy-based criteria (e.g., which aspects of rendering to prioritize) and context-specific knowledge of the application's needs and time budget.

When implementing the adaptive cohort size mechanism, I had several options for

how to manage the previously constant-size scattering and color data buffers that are passed between compute shaders. These options included allocating buffers based on the maximum cohort size allowed, and keeping them throughout the process; allocating buffers at the minimum size and increasing them when needed; overallocating by a fixed amount or factor, to permit some future growth before needing to reallocate; and dynamically reallocating the buffers each time the cohort size changed. I did not exhaustively examine all these options, as they span a range of tradeoffs in memory use, computation time, and code complexity and anyone using the ComputeGlory system should investigate the most effective tradeoff for their own system and performance requirements. I chose to use the last variation. Although the overall elapsed time increases because of the time required to delete and reallocate the buffers at every change, minimizing the data buffer size keeps the shaders' memory usage as small as possible, potentially enabling greater parallelism.

Figure 7.4 shows a trace of adaptive cohort sizing in a rendering scenario where the time budget (solid green bars) alternates between 50ms and 25ms every 20 frames. The host application evaluated past scattering shader performance (orange line) over a three-frame moving window and sized the cohort (blue line) according to the average performance and the current time budget.

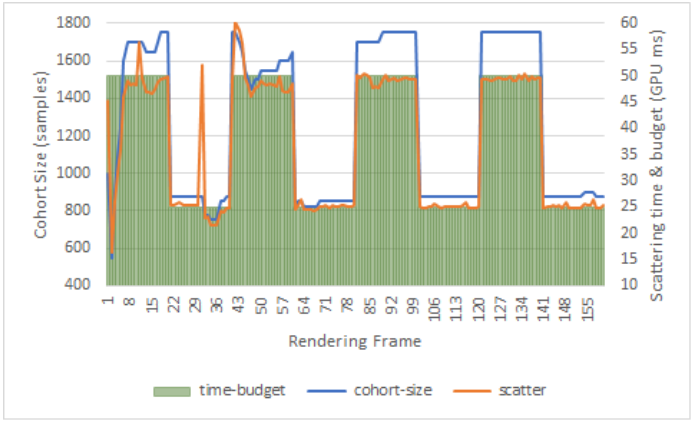


Figure 7.4: Adaptive cohort size responding to a time budget which varies between 20 and 40 ms. Initial cohort size is 5000 samples.

7.4 Timing results

Table 7.4 compares shader timing for k cohorts of n samples vs. a single cohort of $k * n$ samples. The total number of samples computed in all cases is the same: 200,000 samples selected by the 2D Sobol method. k, the number of cohorts, is either 20 or 40, and n (cohort size) is 10,000 or 5000, respectively. The first line of the table shows the GPU time consumed by each shader phase of a single-frame rendering using 200,000 samples. The second line shows average time per frame consumed by an equivalent rendering using a constant cohort size of 5000 samples per frame, and the third line is the total time consumed (over 40 frames) by the same incremental rendering. The incremental rendering does not quite achieve the theoretical 40x speedup but it demonstrates the overwhelming benefit of reducing the number of scattering calculations in a frame, compared to the trivial cost of combining multi-frame results. The fourth and fifth lines show average and total times for cohorts of 10,000 samples, twice the size of the preceding rendering.

I included the timing results for all stages of my GPU shader system, although only the scatter, color-convert, and accumulate shaders are affected by cohort size. The remainder vary with image size and radial slice resolution, which were held constant at 512x512 and 4x in this test. Cohort size was held constant for each case; that is, adaptive cohort sizing

Cohort size	scatter	color	accum	sweep	blend	buf2txt
Single (200K)	5645.45	699.24	n/a	1.86	0.74	0.16
5000 samples total	5700.82	692.62	2.26	74.82	29.786	6.35
5000 samples average	142.52	17.32	0.06	1.87	0.745	0.16
10000 samples total	5911.86	693.00	1.59	37.45	14.93	3.19
10000 samples average	295.59	34.65	0.08	1.87	0.75	0.16

Table 7.4: GPU shader timing (ms) for single frame vs. constant-sized cohorts.

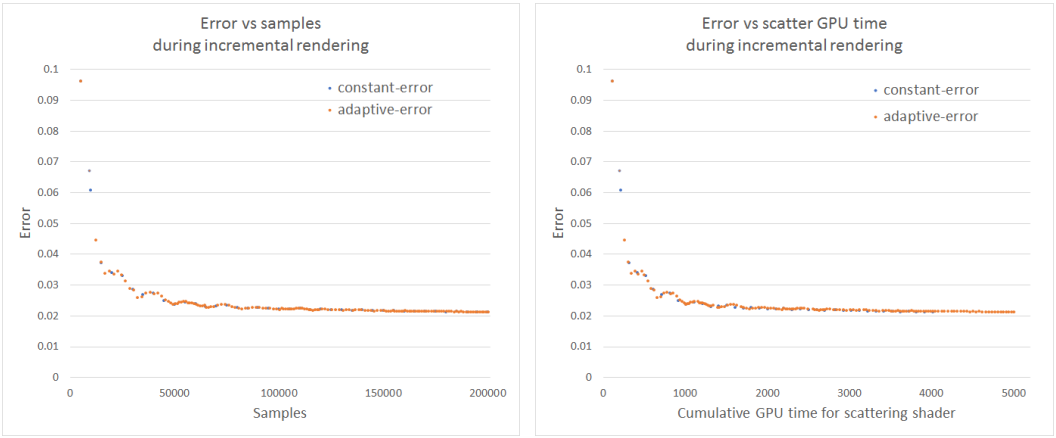
was not enabled for this comparison.

The total scatter shader time consumed by all three cases was approximately the same, with the single-frame version completing fastest and the larger cohort size requiring about 4.7% longer. The total color-conversion times for the two cohort sizes were almost equal, slightly better than the single-frame case. Average per- frame time consumption by the two cohort cases demonstrated the expected factor-of- two difference for both scatter and color-conversion.

I did not exhaustively investigate the ideal cohort size, as in practice that value will vary with other application-specific context and cannot be determined solely from the compute shaders.

7.5 Image quality results

Figure 7.5(a) shows the error convergence for both constant-sized sample cohorts and adaptive cohorts as a function of the number of Mie samples computed. Figure 7.5(b) shows error vs. GPU time expended in the scattering shaders for the same rendering.

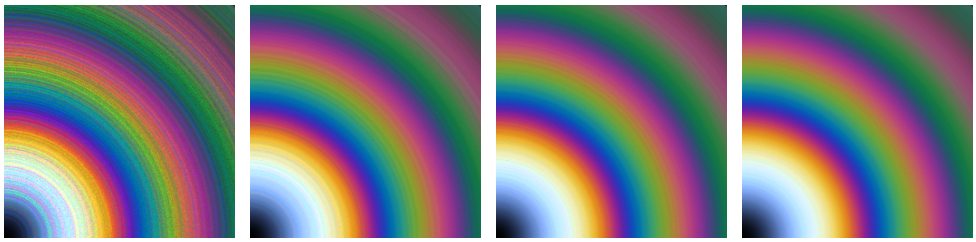


(a) Error vs. Mie calculations (samples). (b) Error vs. scattering shader time.

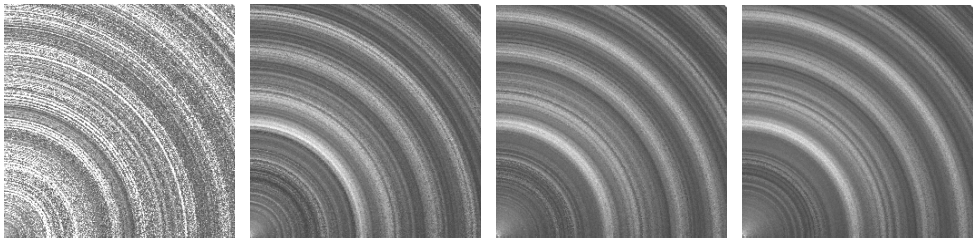
Figure 7.5: Error convergence for constant and adaptive cohort sizes.

Figure 7.6 shows selected frames from an incremental rendering, demonstrating how the results converge as more cohorts are computed. The images in the top row are final

output from the rendering. The bottom row shows the corresponding ΔE^* error images as compared to the reference solution. These images have been brightened and contrast-enhanced for visibility.



(a) Frame 1 (10K samples) (b) Frame 5 (50K) (c) Frame 10 (100K) (d) Frame 20 (200K)



(e) Frame 1 (10K samples) (f) Frame 5 (50K) (g) Frame 10 (100K) (h) Frame 20 (200K)

Figure 7.6: Convergence of glory renderings as more Mie scattering calculations are performed. Top row: final images. Bottom row: ΔE^* error images. (Northeast quadrants)

Chapter 8

CONCLUSION

8.1 Summary

I have developed a GPU-based parallel application to render atmospheric glories at rates significantly faster than previous methods. Rendering times approach real-time for some image sizes and parameter choices on current graphics hardware, and can reasonably be expected to improve with future GPU generations.

The methods used to achieve this speedup include a straightforward transition from a serial CPU-based Mie scattering algorithm to a parallel GPU-based version; task-specific selection of the fastest stable scattering algorithm; image-related optimizations and use of multidimensional low-discrepancy sequences to reduce the number of scattering calculations required; and multi-frame amortization of the scattering calculations.

The rendering and testing framework application presented here for working with glories can also be used for exploring Mie scattering results for other atmospheric phenomena with similar characteristics and physical criteria.

8.2 Contributions

The research contributions of this work include a framework for calculating and rendering wavelength-dependent atmospheric phenomena in real-time by the use of more sophisticated rendering techniques than previously used by scientists researching these phenomena, in combination with parallel GPU techniques for speed and efficiency. The GPU parallel restructuring of the scattering equations presents on its own a significant improvement over existing methods.

I have also developed OpenGL compute shader implementations of several key components which can be used as-is or incorporated into other applications:

- upward- and downward-recurrence Mie scattering
- a fast approximation function for glory rendering
- multiple techniques for generating and using the Sobol sequence and the Tiny Encryption Algorithm
- conversion from wavelength and intensity values to XYZ, LAB, and RGB
- calculation of ΔE^* difference between images

In addition to developing the application used throughout this work, I investigated the benefits of selecting both wavelength and scattering angle as coordinated dimensions of a single quasirandom sequence, rather than choosing angles based on image parameters and selecting wavelengths for each angle independently. This is an example of more general method of recasting a problem from repeated, independent low-dimensional sampling to a single sample in a larger number of correlated dimensions.

I examined the tradeoffs of computation time and convergence behavior of different sampling methods, comparing equal spacing, TEA, and Sobol. Similar analysis would be appropriate in any application domain where sampling is employed.

I illustrated several aspects of incremental computation, including an assessment of the incremental and overall tradeoffs of dividing a large number of Mie calculations into cohorts spread across multiple frames. Adaptive cohort sizing, the use of an approximation as the base of an incremental result, and analysis of the savings achieved by using an approximation are demonstrations of general techniques which are relevant and useful in a wide range of contexts other than atmospheric glory simulation.

I compared timing results for a range of index and table sizes for Sobol sequence partial-table implementation. These results can guide index and table size selection for any applica-

tion of the Sobol partial-table technique, and the methods employed would be appropriate for a more general question of determining the best subdivision scheme for a large problem.

I also performed a timing comparison of two variants of the partial-table technique which are unique to the GPU implementation, and demonstrated that both outperform an equivalent CPU implementation.

The techniques presented here can be useful for both entertainment and scientific applications. Rendering realistic effects at interactive rates supports a greater sense of presence and immersion in simulated environments. Rendering high-accuracy effects at greatly improved speeds (even if not interactive) and rapidly evaluating them against known standards enables physicists to explore and investigate these effects with less delay.

8.3 Evaluation

Through the use of image metrics and evaluation techniques discussed in Chapter 3, I have validated the newly-developed rendering techniques to determine their use and applicability in both entertainment and scientific efforts.

The timing results presented in this work were largely obtained on a mid-grade laptop purchased in 2014. This was not considered a high-end graphics workstation or gaming machine at the time of purchase, and certainly the cutting edge has moved onward during the timespan of this work. Recent GPUs feature greatly increased computational capacity and different balances of floating point vs. integer operations (see Appendix A). These changes offer opportunities to tailor shader code and application structure to maximize hardware support. Additionally, timing was done via a research framework, not a tailored and streamlined interactive application. Finally, the CPU implementation used in Section 4.1 was not optimized to make the best use of modern multi-core CPUs. For all of these reasons, the timing results presented here should be regarded as illustrations of the speedup achieved by the techniques presented, not as peak possible performance for specialized applications on current best-in-class hardware. The results in Appendix A give comparisons for more

recent, more powerful GPUs, and extrapolating from the results obtained on those systems provides basis for optimism about future glory rendering performance.

8.4 Potential extensions and future work

I have identified several optional but highly desirable extensions which can be added to the core system.

- **Droplet size distribution**. IRIS and MiePlot are the only published prior work which explicitly addresses size distribution as a continuous function. Other work, such as Sadeghi et al. (2012), has modelled polydispersion simply by running a complete simulation at each of several discrete sizes and combining the results. Recent preliminary results by Lee and Pahissa (personal communication, 7 September 2018) indicate that integrating a drop-size distribution function over a range of drop sizes reduces the number of wavelengths required per pixel from 500-600 to approximately 300. The methods presented in this dissertation already yield perceptually-acceptable images with sample counts well under 300 wavelengths/pixel, so combining my techniques with Lee and Pahissa's may reduce the required sample count even further. However, if the droplet size or droplet size distribution varies spatially (e.g. from the top to bottom of a cloud, as observed in nature), the radial symmetry optimizations discussed in Chapter 4.2 may no longer be applicable.

- **3D sampling** To implement polydispersion, droplet size (r) could be incorporated into ComputeGlory as a third correlated sampling dimension. Rather than integrating over an independently-determined collection of droplet sizes for each (λ, θ) sample in an offline preliminary step, this method would carry out on-the-fly Monte Carlo integration using samples which are well-distributed in the three-dimensional space (λ, θ, r). A reduction in required sample count appears likely, analogous to the reduc-

tion seen in Chapter 6 when the iterated 1D wavelength selection process was recast as a 2D sampling process.

- **Increased droplet size range**. It should be possible to vary droplet size all the way from glory (4-25 μm) to rainbow (100s of mm) in the same application. It would not be appropriate to display the entire range at once, but being able to explore that range within a single application seems useful.

- **Varying index of refraction**. Index of refraction is a complex-valued parameter to Mie scattering, and its value varies with the wavelength of light being considered, but for this work I have assumed it to be real-valued and constant – specifically, set to 1.33, which lies within the small range observed for water with wavelengths in the visible spectrum. This restriction is appropriate for rendering glories as typically observed in Earth's atmosphere (Laven 2008a). Enabling this value to vary will enable simulation of scenarios outside normal experience, such as the recently observed glory in the atmosphere of Venus (Markiewicz et al. 2014).

I have also identified possible next steps for improving the usefulness of ComputeGlory application or a follow-on application.

- **Volumetric clouds**. Atmospheric phenomena are not found on their own. They require a cloud of droplets in order to be seen, and in turn, they are obscured by multiply-reflected light from the cloud. The visual result in nature is quite different from a raw rendered image. Combining a glory rendering with a volumetrically rendered cloud provides a realistic result. Animating the cloud with a volumetric simulation enhances the realism even further. If the user can interact with the cloud – disrupt the simulation – the experience is even better.

- **Interactive virtual cloud environment**. Carrying this idea further, fast glory rendering supports the creation of a virtual reality environment in which the user can view and interact with clouds that host glories, coronas, and other atmospheric phenomena.

- **User guidance for expectations management**. In a research setting where very-high-resolution glory renderings are desired, the system could provide guidance to the user before beginning a lengthy rendering as to the expected duration of the rendering process, based on previous timing measurements on the same system.

- **Broader range of performance measurements**. To support users wishing to render glories on a variety of computer systems, it would be helpful to collect performance measurements on as broad a range of hardware as possible, including multi-GPU systems and GPU clusters. Although I have primarily targeted upper-end consumer hardware for this work, there is much to be learned by assessing how this method performs on higher-capability systems. In particular, while ComputeGlory did not quite achieve real-time performance on the testbed system or the comparison machines that were available for this work, its performance on additional systems would shed light onto how far away that goal lies and in what direction.

- **Learned estimator**. The approximation function developed for ComputeGlory proved useful as a basis for incremental rendering but was unsuccessful as an estimator for importance sampling. Recent advances in deep learning suggest that a better estimator might be obtained by running numerous renderings and using the results as input so that the system can learn a corresponding function, and new GPUs have been designed specifically to support artificial intelligence applications (NVIDIA 2017).

- **User testing**. No user studies were conducted. The research described here includes both algorithmic validation, confirming the correctness of the rendered images by comparison to reference solutions; and perceptual evaluation using standard metrics which were initially derived from user studies. Direct user evaluation would further confirm the results obtained using those perceptual metrics.

In addition to the extensions and improvements listed above, which are specific to ComputeGlory or glory rendering, I intend to continue investigating a recently-proposed

technique for generating Sobol sequences, called Reordered Sobol (Olano and Blenkhorn, in preparation), which extends the Sobol matrices with additional index bits and permutes the direction numbers to ensure the grid cells fill evenly. This technique holds promise for reducing or eliminating the fill-order problem observed in Chapter 6.

Appendix A

GPU SPECIFICATIONS

A.1 GPU hardware specifications

Table A.1 compares basic hardware specifications for four GPUs. These are

- the GeForce GT 750M, in a 2014 mid-range gaming laptop used for most of the work in this dissertation (described in Section 3.1).
- the Quadro P4000, in the 2018 high-end desktop machine in the VANGOGH lab.
- the GTX 1070, recommended as the minimum graphics card for use with the HTC Vive Pro virtual reality headset.
- the Quadro GV100, announced by NVIDIA Q1 2018 (NVIDIA 2018c).

While there are, of course, many other factors contributing to overall glory rendering time, these statistics give an idea of the speedup that might be expected from the Mie scattering shader if the same timing tests presented in this dissertation were run on more powerful graphics hardware.

Table A.2 shows a timing comparison of the first two GPUs above, using a stripped-down timing application which makes a large number of calls to ComputeGlory's Mie scat-

	GeForce GT 750M	Quadro P4000	GTX 1070	Quadro GV100
family	Kepler	Pascal	Pascal	Volta
microprocessors	2	14	15	80
CUDA cores	384	1792	1920	5120
memory	2GB	8GB	4GB	16GB

Table A.1: Comparison of hardware specifications for four GPUS (see text for further descriptions).

	GeForce GT 750M	Quadro P4000	speedup
average	1810.136	218.9083	8.3

Table A.2: Timing comparison specs for two GPUS performing almost 3 million Mie scattering calculations. Times given in milliseconds. See text for further descriptions.

tering shader, as other experiments have shown this to be by far the largest consumer of GPU compute time in the overall application. The GT750M timing data was collected on the same laptop used throughout this dissertation (described in Section 3.1). The P4000 timing data was collected on a desktop machine purchased in 2018. This machine's CPU was an Intel Xeon W-2145 3.7GHz/4.5GHz with 8 cores, 11M cache, and 32GB (2x16GB) memory.

The timing application ran 2,897,000 calls to the main Mie scattering function. This number corresponds to the scatter-shader workload for a 1024x1024 image using a 4x resolution radial slice, at 1000 wavelengths per pixel. The shader code and dispatch configuration were identical on both machines. Ten timing runs were performed for each system. On average the P4000 completed in 218.9ms, the GT750M in 1810.1ms, giving a speedup of approximately 8.3.

A.2 GPU arithmetic operation throughput

Table A.3 shows operation throughput for selected arithmetic operations on the GPU used for this work as compared to the latest announced NVIDIA GPU (NVIDIA 2018a). The values shown in the columns labelled "per SM" are the number of operations per clock cycle per streaming multiprocessor in the indicated GPU. The "total" columns give the combined operations per clock cycle across all multiprocessors for each GPU, and the last column is the ratio of those totals, showing the greater operation throughput of the newer GPU across all listed categories of operations. Both GPUs compared here have a warp size of 32, meaning one instruction corresponds to 32 operations (NVIDIA 2018d). Therefore, the instruction throughput is 1/32 times the operation throughput. The last row of the table

	GT 750M		GV100		ratio
	per SM	total	per SM	total	
32-bit float					
add, multiply, multiply-add	192	384	64	5120	13.3
sqrt, sine, cosine	32	64	16	1280	20
32-bit integer					
add/subtract	160	320	64	5120	16
multiply	32	64	64	5120	80
32-bit bitwise AND, OR, XOR	160	320	64	5120	16
64-bit float					
add, multiply, multiply-add	8	16	32	2560	160

Table A.3: Throughput (ops per clock cycle) for arithmetic operations on the GT 750M (used for this work), and on the GV100 (2018).

gives the same comparison for 64-bit floating point operations. However, ComputeGlory does not use 64-bit floating point operations in its compute shaders, as OpenGL does not yet provide support for double-precision trigonometric functions (Khronos 2012).

Appendix B

IMAGE CREDITS AND SOURCES

Images not listed here are original work.

2.1(a)	Rainbow	Public domain, courtesy of user anna langova
2.1(b)	Fogbow	By Thoth God of Knowledge, Creative Commons license
2.1(c)	Corona	By Wing-Chi Poon, Own work [CC BY-SA 2.5], via Wikimedia Commons
2.1(d)	Glory	By Sandstein, Own work [CC BY 3.0], via Wikimedia Commons
2.2(a)	Rainbow schematic	By Rene Descartes 1637. Public domain via Wikimedia Commons
2.2(b)	Rainbow refr./refl.	By KES47, Own work Public domain via Wikimedia Commons
2.4	Lee diagram	modified from Lee (1998) with permission
2.3(a)	CIE RGB	Public domain, courtesy of user Quibik via Wikipedia
2.3(b)	CIE XYZ	SVG file under GDFL (http://fsf.org) by user Acdx via Wikipedia, based on Wyman, Sloan, and Shirley (2013)

REFERENCES

Airy, G. B. 1838. "On the intensity of light in the neighbourhood of a caustic." *Trans. Cambridge Philos. Soc.* 6:379–403.

Antonov, Ilya A, and VM Saleev. 1979. "An economic method of computing LP τ-sequences." *USSR Computational Mathematics and Mathematical Physics* 19 (1): 252–256.

Baek, Jongmin, and David E. Jacobs. 2010. "Accelerating Spatially Varying Gaussian Filters." In *ACM SIGGRAPH Asia 2010 Papers,* 169:1–169:10. SIGGRAPH ASIA '10. Seoul, South Korea: ACM. isbn: 978-1-4503-0439-9. doi:10.1145/1866158.1866191. http://doi.acm.org/10.1145/1866158.1866191.

Balzer, Michael, Thomas Schlömer, and Oliver Deussen. 2009. "Capacity-constrained Point Distributions: A Variant of Lloyd's Method." *ACM Trans. Graph.* (New York, NY, USA) 28, no. 3 (July): 86:1–86:8. issn: 0730-0301. doi:10.1145/1531326.1531392.

Baraff, David, and Andrew Witkin. 1998. "Large Steps in Cloth Simulation." In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques,* 43–54. SIGGRAPH '98. New York, NY, USA: ACM. isbn: 0-89791-999-8. doi:10.1145/280814.280821. http://doi.acm.org/10.1145/280814.280821.

Bohren, Craig F, and Donald R Huffman. 1983. *Absorption and scattering of light by small particles.* John Wiley & Sons.

Box, G.E.P., and M.E. Müller. 1958. "A note on the generation of random normal deviates." *Ann. Math. Statist.* 29:610–611.

Brandt, R. K., and R. G. Greenler. 2001. “Color simulation of size-dependent features of rainbows.” Seventh Topical Meeting on Meteorological Optics, Boulder, Colorado, 5-8 June.

Bratley, Paul, and Bennett L. Fox. 1988. “Algorithm 659: Implementing Sobol’s Quasirandom Sequence Generator.” *ACM Trans. Math. Softw.* (New York, NY, USA) 14, no. 1 (March): 88–100. issn: 0098-3500. doi:10.1145/42288.214372.

Brewer, Clint. 2004. “Rainbows and Fogbows: Adding Natural Phenomena.” *NVIDIA Corporation, SDK white paper.* http://download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/HLSL_RainbowFogbow/docs/RainbowFogbow.pdf.

Bridson, Robert, Ronald Fedkiw, and John Anderson. 2005. “Robust Treatment of Collisions, Contact and Friction for Cloth Animation.” In *ACM SIGGRAPH 2005 Courses.* SIGGRAPH ’05. Los Angeles, California: ACM. doi:10.1145/1198555.1198572. http://doi.acm.org/10.1145/1198555.1198572.

Bruton, D. 1996. *Color science.* http://www.physics.sfasu.edu/astro/color/spectra.html.

Chiu, Kenneth, Peter Shirley, and Changyaw Wang. 1994. “Graphics Gems IV.” Chap. Multi-jittered Sampling, edited by Paul S. Heckbert, 370–374. San Diego, CA, USA: Academic Press Professional, Inc. isbn: 0-12-336155-9.

CIE. 1931. *Commission internationale de l’Eclairage proceedings.* Cambridge University.

———. 1976. *CIE L*a*b**. ISO 11664-4:2008(E)/CIE S 014-4/E:2007.

Commission Internationale de L’eclairage. 1995. *CIE 116-1995, Industrial Colour-Difference Evaluation.* http://www.cie.co.at.

Cook, Robert L. 1986. "Stochastic sampling in computer graphics." *ACM Transactions on Graphics (TOG)* 5 (1): 51–72.

Cook, Robert L., Thomas Porter, and Loren Carpenter. 1984. "Distributed Ray Tracing." In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques,* 137–145. SIGGRAPH '84. New York, NY, USA: ACM. isbn: 0-89791-138-5. doi:10.1145/800031.808590.

Cowley, L. *IRIS*. http://www.atoptics.co.uk/droplets/iris.htm.

———. 1998. *Atmospheric Optics (website).* Technical report. http://www.atoptics.co.uk.

Cowley, Les, Philip Laven, and Michael Vollmer. 2005. "Rings around the sun and moon: coronae and diffraction." *Physics education* 40 (1): 51.

Cowley, Les, and Michael Schroeder. *AirySim*. http://www.atoptics.co.uk/rainbows/airysim.htm.

———. *BowSim*. http://www.atoptics.co.uk/rainbows/bowsim.htm.

———. *HaloSim3*. http://www.atoptics.co.uk/halo/halfeat.htm.

Dave, J. V. 1968. *Subroutines for Computing the Parameters of the Electromagnetic Radiation Scattered by a Sphere*. IBM Order Number 360D-17.4.002.

Debye, P. 1908. "Das Elektromagnetische Feld um einen Zylinder und die Theorie des Regenbogens." *Phys. Z.* 9:775–778.

———. 1944. "Light scattering in solutions." *J. Appl. Phys.* 15:338–342.

Dippé, Mark AZ, and Erling Henry Wold. 1985. "Antialiasing through stochastic sampling." *ACM Siggraph Computer Graphics* 19 (3): 69–78.

Dobkin, David P., David Eppstein, and Don P. Mitchell. 1996. "Computing the Discrepancy with Applications to Supersampling Patterns." *ACM Trans. Graph.* (New York, NY, USA) 15, no. 4 (October): 354–376. issn: 0730-0301. doi:10.1145/234535.234536. http://doi.acm.org/10.1145/234535.234536.

Facebook Technologies, LLC. 2018. *Oculus Rift product page*. Retrieved 4 February 2018. https://www.oculus.com/rift/.

Fairchild, Mark D. 2005. *Color Appearance Models.* 2nd ed. West Sussex, England: John Wiley & Sons. isbn: 978-0470012161.

Faure, H. 1992. "Good permutations for extreme discrepancy." *Journal of Number Theory* 42 (1): 47–56.

Faure, Henri. 1982. "Discrépance de suites associées à un système de numération (en dimension s)." *Acta Arithmetica* 41:337–351.

Fei, Yun (Raymond), Christopher Batty, Eitan Grinspun, and Changxi Zheng. 2018. "A Multi-scale Model for Simulating Liquid-fabric Interactions." *ACM Trans. Graph.* (New York, NY, USA) 37, no. 4 (July): 51:1–51:16. issn: 0730-0301. doi:10.1145/3197517.3201392. http://doi.acm.org/10.1145/3197517.3201392.

Flautau, Piotr. 1998. *bhmie-c.* http://scatterlib.wikidot.com/mie.

Galanti, S., and A. Jung. 1997. "Low-discrepancy sequences: Monte Carlo simulation of option prices." *Journal of Derivatives* 5 (1): 63–83.

Gedzelman, Stanley D., and James A. Lock. 2003. "Simulating coronas in color." *Appl. Opt.* 42, no. 3 (January): 497–504. doi:10.1364/AO.42.000497. http://ao.osa.org/abstract.cfm?URI=ao-42-3-497.

Georgiev, Iliyan, and Marcos Fajardo. 2016. "Blue-noise Dithered Sampling." In *ACM SIGGRAPH 2016 Talks,* 35:1–35:1. SIGGRAPH '16. Anaheim, California: ACM. isbn: 978-1-4503-4282-7. doi:10.1145/2897839.2927430.

Greenler, Robert. 1980. *Rainbows, Halos, and Glories.* Cambridge University Press.

Grünschloß, Leonhard, Matthias Raab, and Alexander Keller. 2012. "Enumerating quasi-monte carlo point sequences in elementary intervals." In *Monte Carlo and Quasi-Monte Carlo Methods 2010,* 399–408. Springer.

Halton, J. H. 1964. "Algorithm 247: Radical-inverse Quasi-random Point Sequence." *Commun. ACM* (New York, NY, USA) 7, no. 12 (December): 701–702. issn: 0001-0782. doi:10.1145/355588.365104.

Hammersley, J. M., and K.W. Morton. 1954. "Poor Man's Monte Carlo." *Journal of the Royal Statistical Society, Series B (Methodological)* 16 (1): 23–38.

Hammersley, John M. 1960. "Monte Carlo methods for solving multivariable problems." *Annals of the New York Academy of Sciences* 86 (3): 844–874.

Harder, J. W., G. Lawrence, J. Fontenla, G.J. Rottman, and T.N. Woods. 2005. "The Spectral Irradiance Monitor: Scientific requirements, instrument design, and operation modes." *Solar Phys.* 230:141–167.

HTC Corporation. 2018. *HTC Vive product page.* Retrieved 4 February 2018. https://www.vive.com/us/product/vive-virtual-reality-system/.

Joe, Stephen, and Frances Y. Kuo. 2008. "Constructing Sobol Sequences with Better Two-Dimensional Projections." *SIAM J. Sci. Comput.* (Philadelphia, PA, USA) 30, no. 5 (August): 2635–2654. issn: 1064-8275. doi:10.1137/070709359.

Kensler, Andrew. 2016. *Correlated Multi-jittered Sampling.* Technical report Technical Memo 13-01. Pixar, March.

Khronos. 2012. *ARB_gpu_shader_fp64 specification.* Retrieved 1 Oct 2018. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_gpu_shader_fp64.txt.

Kolasinski, Eugenia M. 1995. *Simulator Sickness in Virtual Environments.* Technical report. Army Research Institute for the behavioral and social sciences Alexandria VA.

Kollig, Thomas, and Alexander Keller. 2002. "Efficient multidimensional sampling." In *Computer Graphics Forum,* 21:557–563. 3. Wiley Online Library.

Lagae, Ares, and Philip Dutré. 2008. "A comparison of methods for generating Poisson disk distributions." In *Computer Graphics Forum,* 27:114–129. 1. Wiley Online Library.

Laven, Philip. *MiePlot.* http://www.philiplaven.com/mieplot.htm.

———. 2003. "Simulation of rainbows, coronas, and glories by use of Mie theory." *Appl. Opt.* 42 (3): 436–444.

———. 2005. "Atmospheric glories: simulations and observations." *Appl. Opt.* 44 (27): 5667–5674.

———. 2006. *Understanding Glories: Introduction to Mie Theory,* July. http://www.philiplaven.com/p2c1.html.

———. 2008a. "Effects of refractive index on glories." *Appl. Opt.* 47, no. 34 (December): H133–H142. doi:10.1364/AO.47.00H133. http://ao.osa.org/abstract.cfm?URI=ao-47-34-H133.

———. 2008b. *How are glories formed?,* July. http://www.philiplaven.com/p2c1a.html.

LaViola, Joseph J., Jr. 2000. "A Discussion of Cybersickness in Virtual Environments." *SIGCHI Bull.* (New York, NY, USA) 32, no. 1 (January): 47–56. issn: 0736-6906. doi:10.1145/333329.333344. http://doi.acm.org/10.1145/333329.333344.

Lee, Raymond L. 1998. “Mie theory, Airy theory, and the natural rainbow.” *Appl. Opt.* 37 (9): 1506–1519.

Lindbloom, Bruce. 2015. *Delta E (CIE 1994)*. http://www.brucelindbloom.com/index.html?Eqn_DeltaE_CIE94.html.

———. 2017a. *RGB/XYZ Matrices*. http://www.brucelindbloom.com/index.html?Eqn_RGB_to_XYZ.html.

———. 2017b. *XYZ to Lab*. http://www.brucelindbloom.com/index.html?Eqn_XYZ_to_Lab.html.

Lloyd, S. 1982. “Least squares quantization in PCM.” *IEEE Transactions on Information Theory* 28, no. 2 (March): 129–137. issn: 0018-9448. doi:10.1109/TIT.1982.1056489.

Markiewicz, W.J., E. Petrova, O. Shalygina, M. Almeida, D.V. Titov, S.S. Limaye, N. Ignatiev, T. Roatsch, and K.D. Matz. 2014. “Glory on Venus cloud tops and the unknown UV absorber.” *Icarus* 234:200–203. issn: 0019-1035. doi:http://dx.doi.org/10.1016/j.icarus.2014.01.030. http://www.sciencedirect.com/science/article/pii/S001910351400061X.

Maxwell, James Clerk. 1873. *A Treatise on Electricity and Magnetism*. Oxford: Clarendon Press.

Mearian, Lucas. 2018. “Bitcoin mining leads to an unexpected GPU gold rush” (April 2). https://www.computerworld.com/article/3267744/computer-hardware/bitcoin-mining-leads-to-an-unexpected-gpu-gold-rush.html.

Microsoft. 2018. *<chrono> header documentation, Microsoft Developers Network*. Retrieved 8 February 2018. https://msdn.microsoft.com/en-us/library/hh874757.aspx.

Mie, Gustav. 1908. “Beiträge zur Optik trüber Medien, speziell kolloidaler Metallösungen.” *Annalen der Physik* 330 (3): 377–445. issn: 1521-3889. doi:10.1002/andp.19083300302. http://dx.doi.org/10.1002/andp.19083300302.

Nussenzveig, H. Moyses. 2012. “The Science of the Glory.” *Scientific American* 306 (1): 68–73. http://www.scientificamerican.com/article/the-science-of-the-glory/.

Nussenzveig, H.M. 2002. “Does the glory have a simple explanation?” *Opt. Lett.* 27, no. 16 (August): 1379–1381. doi:10.1364/OL.27.001379. http://ol.osa.org/abstract.cfm?URI=ol-27-16-1379.

NVIDIA. 2012. *Tesla K20 GPU Accelerator Board Specification.* Retrieved 4 February 2018. https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf.

———. 2017. *NVIDIA Tesla V100 GPU Architecture.* Retrieved 1 March 2018. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

———. 2018a. *CUDA C Programming Guide v10.0.130.* Retrieved 21 March 2018. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions.

———. 2018b. *GeForce 750M Specifications.* Retrieved 4 February 2018. https://www.geforce.com/hardware/notebook-gpus/geforce-gt-750m/specifications.

———. 2018c. *GV100 information sheet.* Retrieved 1 Oct 2018. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/documents/quadro-volta-gv100-us-nv-623049-r10-hr.pdf.

NVIDIA. 2018d. *NVIDIA CUDA toolkit documentation.* Retrieved 1 Oct 2018. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities.

Olano, Marc, and Ari Blenkhorn. in preparation. "Cell-Indexed and Reordered Sobol Sequences."

Owen, Art B. 1995. "Randomly Permuted (t,m,s)-Nets and (t, s)-Sequences." In *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing: Proceedings of a conference at the University of Nevada, Las Vegas, Nevada, USA, June 23–25, 1994,* edited by Harald Niederreiter and Peter Jau-Shyong Shiue, 299–317. New York, NY: Springer New York. isbn: 978-1-4612-2552-2. doi:10.1007/978-1-4612-2552-2_19.

Rapkin, Ari. 2002. "How to Dress Like a Jedi: Techniques for Digital Clothing." In *ACM SIGGRAPH 2002 Conference Abstracts and Applications,* 196–196. SIGGRAPH '02. San Antonio, Texas: ACM. isbn: 1-58113-525-4. doi:10.1145/1242073.1242209. http://doi.acm.org/10.1145/1242073.1242209.

Rapkin, Ari, Andy Anderson, William Clay, John Helms, and Eric Wong. 2003. "Getting Ripped: Hulking Out the Clothing Pipeline for Shredding, Tearing, and Electricity." In *ACM SIGGRAPH 2003 Sketches &Amp; Applications.* SIGGRAPH '03. San Diego, California: ACM.

Ray, B. B. 1923. "The scattering of liquid droplets, and the theory of coronas, glories and iridescent clouds." *Proc. Ind. Assoc. for the Cultiv. of Sci.* 8:23–46.

Riley, Kirk, David S. Ebert, Martin Kraus, Jerry Tessendorf, and Charles Hansen. 2004. "Efficient Rendering of Atmospheric Phenomena." In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques,* 375–386. EGSR'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. isbn: 3-905673-12-6. doi:10.2312/EGWR/EGSR04/375-386. http://dx.doi.org/10.2312/EGWR/EGSR04/375-386.

Robertson, A. R. 1990. "Historical development of CIE recommended color difference equations." *Color Research & Application* 15 (3): 167–170. issn: 1520-6378. doi:10.1002/col.5080150308. http://dx.doi.org/10.1002/col.5080150308.

Rosenbluth, Marshall N, and Arianna W Rosenbluth. 1955. "Monte Carlo calculation of the average extension of molecular chains." *The Journal of Chemical Physics* 23 (2): 356–359.

Sadeghi, Iman, Adolfo Muñoz, Philip Laven, Wojciech Jarosz, Francisco Seron, Diego Gutierrez, and Henrik Wann Jensen. 2012. "Physically-based Simulation of Rainbows." *ACM Transactions on Graphics (Presented at SIGGRAPH)* 31, no. 1 (February): 3:1–3:12. doi:10.1145/2077341.2077344.

Schneider, A. 2015. "The Real-Time Volumetric Cloudscapes Of Horizon: Zero Dawn." In *Advances in Real Time Rendering, Part I, ACM SIGGRAPH 2015 Courses*. Los Angeles, CA: ACM SIGGRAPH. isbn: 978-1-4503-3634-5. doi:10.1145/2776880.2787701. https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn.

Schröder, Wilfried, and Karl-Heinrich Wiederkehr. 2000. "Johann Kiessling, the Krakatoa Event and the Development of Atmospheric Optics after 1883." *Notes and Records of the Royal Society of London* 54 (2): 249–258. issn: 00359149. http://www.jstor.org/stable/531969.

Shen, Jianqi, and Huarui Wang. 2010. "Calculation of Debye series expansion of light scattering." *Appl. Opt.* 49, no. 13 (May): 2422–2428. doi:10.1364/AO.49.002422. http://ao.osa.org/abstract.cfm?URI=ao-49-13-2422.

Shirley, P. 1991. "Discrepancy as a quality measure for sampling distributions," 183–194. Proc. Eurographics '91. September.

Sobol', I. M. 1967. "On the distribution of points in a cube and the approximate evaluation of integrals." Zh. Vȳchisl. Mat. Mat. Fiz., 7 (1967), 784-802 (in Russian), *USSR Computational Mathematics and Mathematical Physics* 7 (4): 86–112. doi:10.1016/0041-5553(67)90144-9.

———. 1976. "Uniformly distributed sequences with an additional uniform property." Zh. Vȳchisl. Mat. Mat. Fiz., 16 (1976), 1332-1337 (in Russian), *USSR Computational Mathematics and Mathematical Physics* 16 (5): 236–242. issn: 0041-5553. doi:10.1016/0041-5553(76)90154-3.

Strutt, John. 1871. "On the scattering of light by small particles." *Philosophical Magazine,* 4th ser., 41:447–454.

The International Association for the Properties of Water and Steam. 1997. *Release on the Refractive Index of Ordinary Water Substance as a Function of Wavelength, Temperature and Pressure.* Retrieved 29 December 2017. http://www.iapws.org/relguide/rindex.pdf.

van de Hulst, H. C. 1958. "Light scattering by small particles." *Quarterly Journal of the Royal Meteorological Society* 84 (360): 198–199. issn: 1477-870X. doi:10.1002/qj.49708436025.

Waterman, P. C. 1965. "Matrix formulation of electromagnetic scattering." *Proceedings of the IEEE* 53:805–812.

Wheeler, David J., and Roger M. Needham. 1995. "TEA, a tiny encryption algorithm." In *Fast Software Encryption,* edited by Bart Preneel, 1008:363–366. Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Wiscombe, W. J. 1980. "Improved Mie scattering algorithms." *Appl. Opt.* 19, no. 9 (May): 1505–1509. doi:10.1364/AO.19.001505. http://ao.osa.org/abstract.cfm?URI=ao-19-9-1505.

Wiscombe, Warren J. 1979. *Mie scattering calculations: advances in technique and fast, vector-speed computer codes.* Technical Note. National Center for Atmospheric Research. Boulder, Colorado.

Wyman, Chris, Peter-Pike Sloan, and Peter Shirley. 2013. "Simple Analytic Approximations to the CIE XYZ Color Matching Functions." *Journal of Computer Graphics Techniques (JCGT)* 2, no. 2 (July 12): 1–11. issn: 2331-7418. http://jcgt.org/published/0002/02/01/.

Wyszecki, Günter, and W.S. Stiles. 1982. *Color Science – Concepts and Methods, Quantitative Data and Formula.* 725–735. John Wiley.

Zafar, Fahad, Marc Olano, and Aaron Curtis. 2010. "GPU Random Numbers via the Tiny Encryption Algorithm." In *Proceedings of the Conference on High Performance Graphics,* 133–141. HPG '10. Saarbrucken, Germany: Eurographics Association. http://dl.acm.org/citation.cfm?id=1921479.1921500.