Adaptive Spectral Mapping for Real-Time Dispersive Refraction

by

Damon Blanchette

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

January 2012

APPROVED:

____

Professor Emmanuel Agu, Thesis Adviser

____

Professor Matthew Ward, Thesis Reader

____

Professor Craig Wills, Head of Department

Abstract

Spectral rendering, or the synthesis of images by taking into account the wavelengths of light, allows effects otherwise impossible with other methods. One of these effects is dispersion, the phenomenon that creates a rainbow when white light shines through a prism. Spectral rendering has previously remained in the realm of off-line rendering (with a few exceptions) due to the extensive computation required to keep track of individual light wavelengths. Caustics, the focusing and de-focusing of light through a refractive medium, can be interpreted as a special case of dispersion where all the wavelengths travel together. This thesis extends Adaptive Caustic Mapping, a previously proposed caustics mapping algorithm, to handle spectral dispersion. Because ACM can display caustics in real-time, it is quite amenable to be extended to handle the more general case of dispersion. A method is presented that runs in screen-space and is fast enough to display plausible dispersion phenomena in real-time at interactive frame rates.

Acknowledgments

I would like to thank my adviser, Professor Emmanuel Agu, for his guidance, laughs, and answering my hundreds of questions over the year it took to complete this thesis. The beautiful video card helped, too. A great thanks goes to Professor Matthew Ward for agreeing to be the reader for this thesis. He was also my MQP adviser when I was an undergrad at WPI, and that experience shaped my future progress towards this thesis.

I would like to thank my son Emmet and my parents, Maureen and Don, for putting up with me while I've been staring closely at my computer monitor and giving me the personal time required to undertake this project.

Lastly, a huge thanks to these musicians: Umphrey's McGee, The Disco Biscuits, Lotus, Clutch, Karma to Burn, and the Psytrance music genre. There are others, but the list is too long to mention them all. I would not have been able to complete this thesis without your fantastic tunes coming out of my speakers every time I sat down in front of my computer.

Table of Contents

Table of Figures

Chapter 1: Introduction

Spectral rendering is the synthesis of images by taking into account the wave properties of light. Underlying mechanisms such as refraction, interference, and diffraction cause white light to be split into its constituent wavelengths, generating iridescent colors. Most graphics techniques and engines only perform lighting calculations utilizing the particle nature of light, which does not give a full picture of the world around us. Images generated with spectral rendering enabled can display such phenomena as the rainbows that occur when white light shines through a prism and the beautiful colors that appear inside gemstones such as diamonds. Figure 1 illustrates these effects.

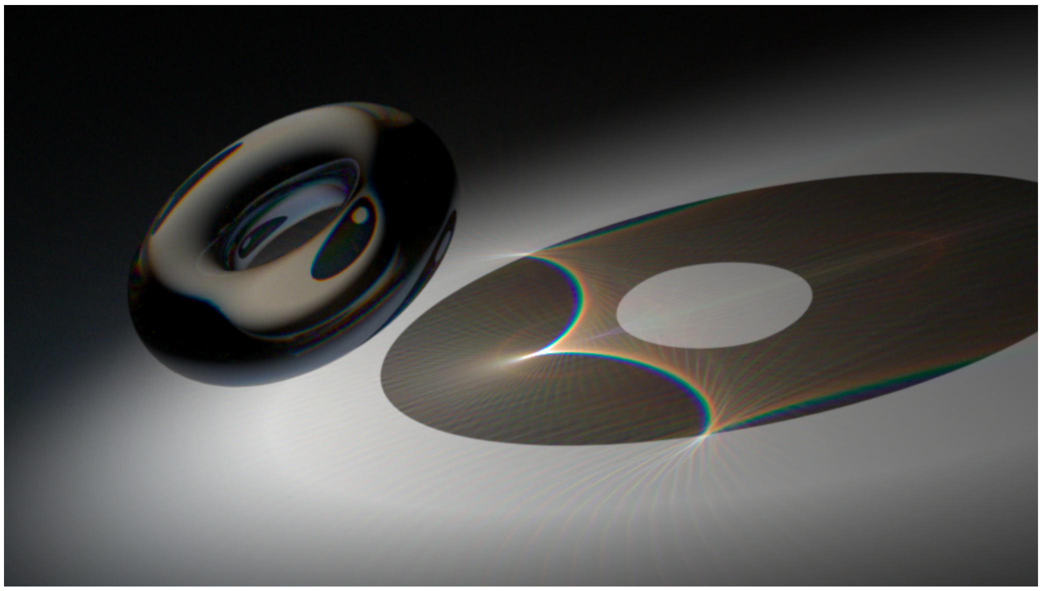


Figure 1: Image generated using spectral rendering with LuxRender [Lux]. Render time was one hour. Dispersion amount has been set very high in order to more effectively show the phenomenon. Note the rainbows on both the "floor" and on the surface of the torus.

In terms of refraction, which is the focus of this thesis, most graphics implementations of any kind (offline or online) treat the refractive index of an object as constant throughout the light spectrum. However, when light travels through a transparent object it is not simply refracted, it is *dispersed* – different wavelengths of light refract at different angles through the object. This thesis presents an alogrithm to simulate and display this important phenomenon, responsible for rainbows in the sky, in real-time on current hardware.

Speed of image synthesis is an issue for spectral renderers. LuxRender [Lux], an open source unbiased offline renderer, for example, uses bidirectional path tracing combined with Metropolis sampling in its spectral rendering engine. This method is physically accurate, but far from amenable to real-time graphics, as evidenced by the 1-hour render time of figure 1 for a noise-free image. To increase speed, the algorithm presented in this thesis makes a number of approximations, but strives to maintain the highest quality final image.

Performance issues arise in spectral rendering for multiple reasons. The most prominent is that complex equations need to be evaluated for multiple wavelengths. To calculate truly accurate physically-based indices of refraction for example, complex numbers need to be utilized. Depending on the chosen method of sampling the spectrum and converting wavelengths to RGB colors, performance can be reduced. Choosing discrete wavelengths may be faster, but evaluating the spectrum using a function or using many wavelength samples can slow performance down greatly. Transitioning from wavelengths to RGB using a CIE XYZ representation of colors and evaluating the conversion functions can be a slow process as well.

1.1 Spectral Phenomena

There are multiple phenomena of light that occur due to its spectral nature. These include dispersion, diffraction, interference, and scattering. In this section an overview of the light spectrum is presented, along with what it means for each of the phenomena mentioned above.

Light itself is composed of a long continuous spectrum, with wavelengths ranging from very large: ~10^3 m for radio waves, to the very small: ~10^{-12} m for gamma rays. Figure 2 is a diagram illustrating the wavelengths of light and where the visible spectrum fits in.

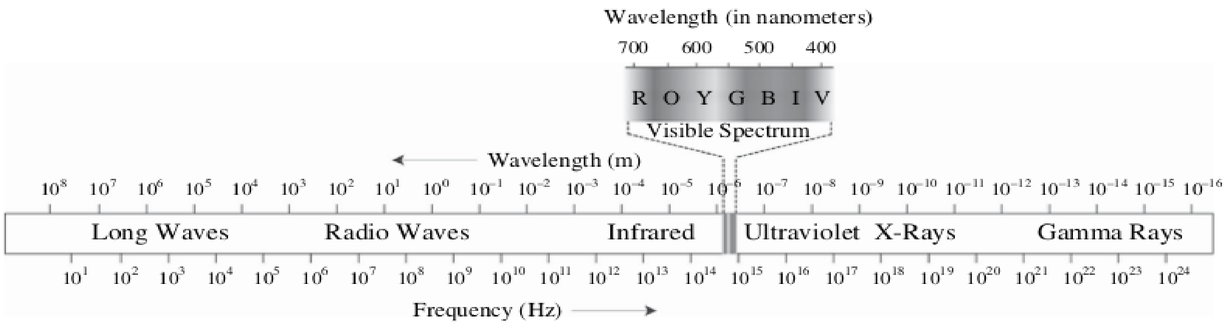


Figure 2: The electromagnetic spectrum, with the visible spectrum highlighted. From [DD08].

The visible spectrum is centered around what the human eye can see, which ranges from around 400 nm (violet) wavelength to around 700 nm (red). Figure 3 is an image of the full visible spectrum, from a photograph taken by the author of dispersed light out of a prism.

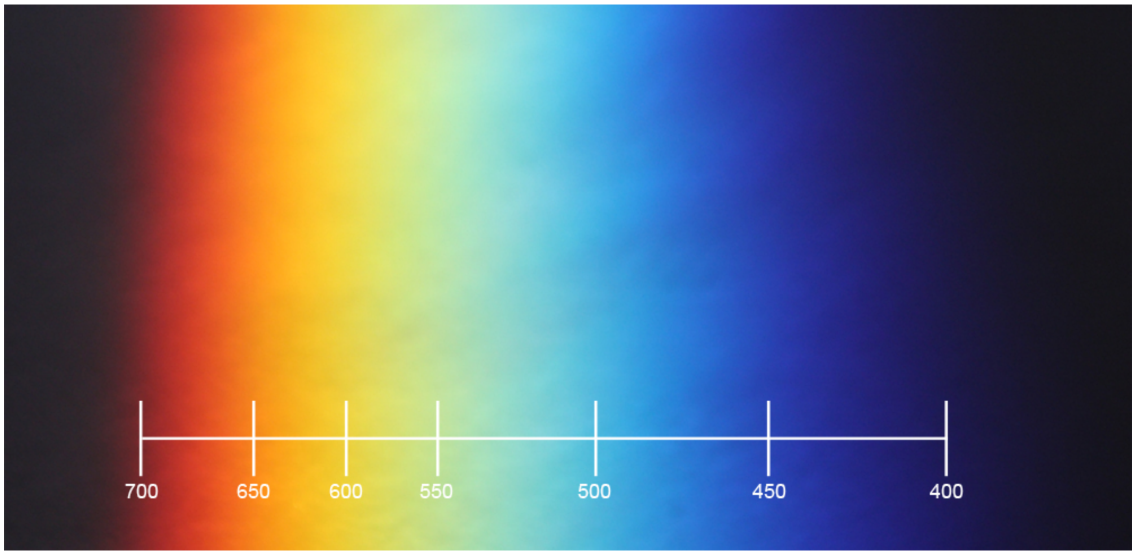


Figure 3: The visible spectrum with wavelengths labeled, from a photograph by the author. Units are in nanometers. The scale is squashed and stretched in places because of the physical size of the color bands created from the prism that dispersed the light.

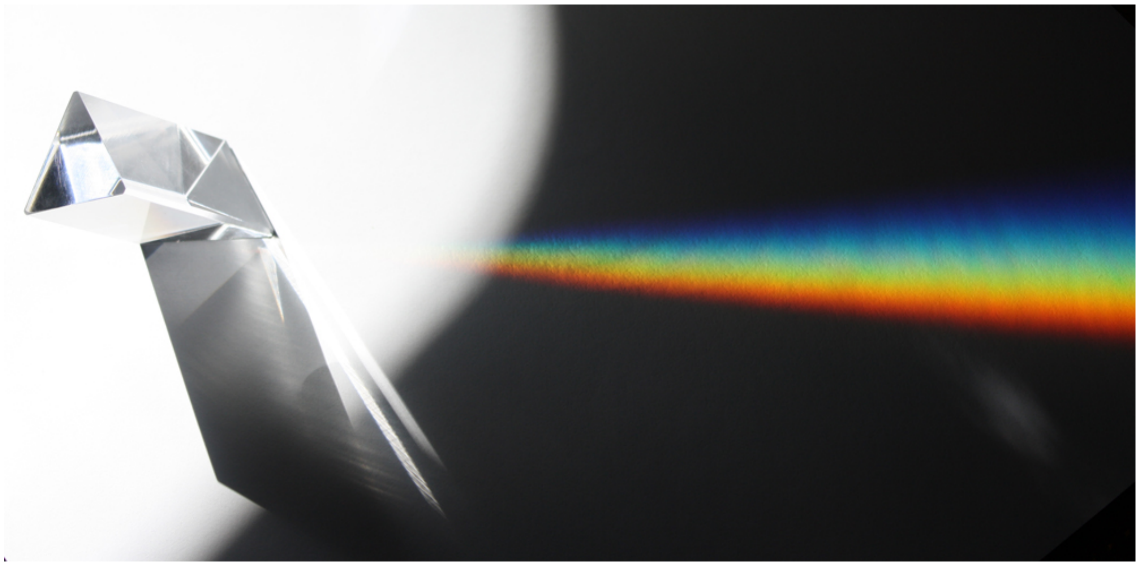


Figure 4: Spectral dispersion through an acrylic prism. Photograph by the author.

Dispersion is a phenomenon that occurs because light of different wavelengths refracts at different angles. A rainbow is one special case of dispersion; in fact it occurs throughout the entire electromagnetic spectrum, and in other kinds of waves as well, such as those of sound. Figure 4 shows an example photograph of visible light dispersion through an acrylic prism.

The amount of refraction varies by wavelength depending on the material the light is traveling through. A material such as a diamond, with a high refractive index, disperses light far more noticeably than a material such as glass, which has a lower refractive index. Even air disperses light, though in amounts so miniscule it is not normally noticeable by the human eye.

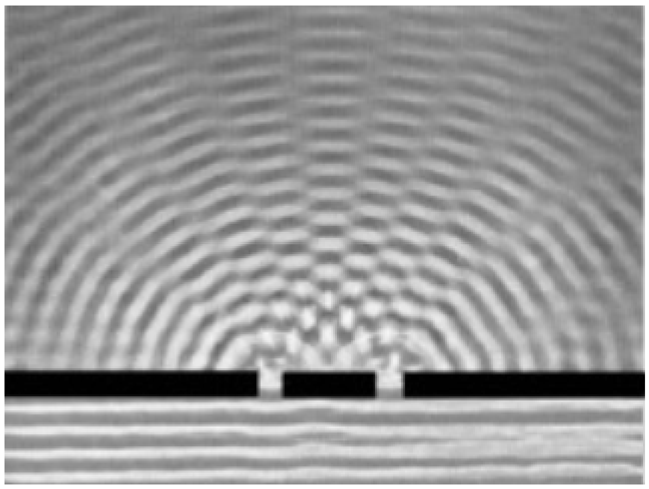


Figure 5: Diffraction in sinusoidal water waves, which are coming up from the bottom and encountering a wall with two slits. From [Cro10].

A second phenomena that separates white light into its component wavelengths is diffraction. Diffraction can be described as the behavior of a wave when it encounters an obstacle or a non-uniformity in its medium [Cro10]. Like dispersion, diffraction works not just with visible light, but occurs in the entire electromagnetic spectrum and other waves as well. Figure 5 shows what happens when a water wave encounters a barrier with two gaps in it. The wave nature of light was in fact originally recognized when François Arago and Augustin-Jean Fresnel were performing their experiments in light diffraction [Cha05]. In terms of light, you can see diffraction yourself when looking at the back of a CD, where the familiar rainbow patterns are due to the individual tracks acting as a diffraction grating. Figure 6 shows this phenomenon on a CD, and Figure 7 shows diffraction caused by tiny scratches and possibly from the manufacturing process of an acrylic prism.

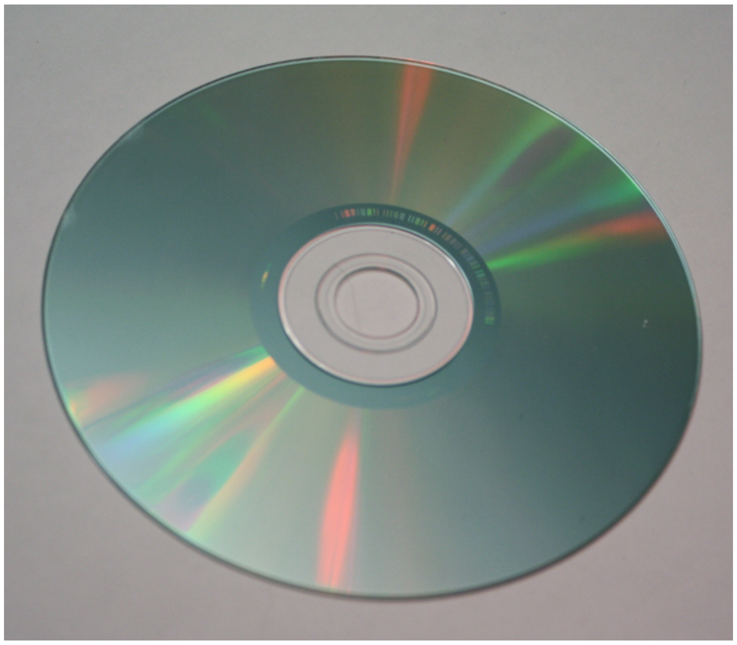


Figure 6: An image of the back of a compact disc demonstrating optical diffraction. Photo taken by author.

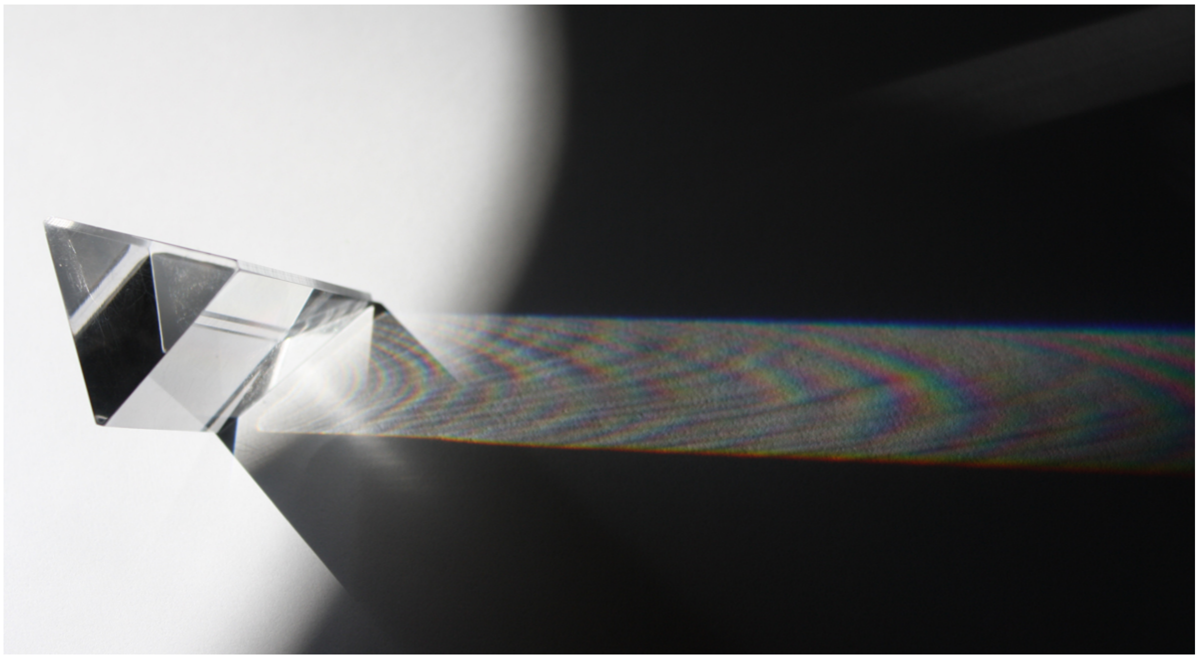


Figure 7: Diffraction due to light waves being blocked by tiny scratches and a pattern likely caused by the manufacturing process of this prism. Photo taken by the author.

A phenomenon reminiscent of diffraction that creates similar effects is interference. In terms of light waves, the most familiar form of interference is called thin-film interference. This phenomenon is what creates the rainbow patterns seen in oil slicks or on bubbles. In essence, it occurs when light waves reflected by the upper and lower boundaries of a thin film (such as oil on a road or a bubble's surface) interfere with each other. Figure 8 shows a synthesized image example of thin film interference.

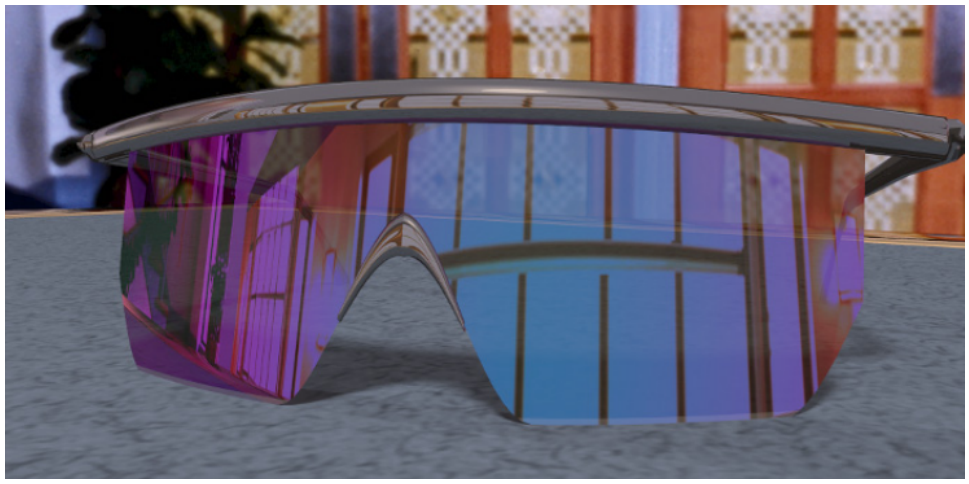


Figure 8: Sunglasses with a thin film coating. Image from [GMN94]

Scattering is a phenomenon that also occurs due to the wave nature of light. Light scatters when certain wavelengths are affected by a medium and others are not. The scattered wavelengths are the color we see. It, along with absorption, is the reason objects in the world are observed to be a certain color. Scattering, specifically Rayleigh scattering, is the reason that the sky is blue: the composition of the atmosphere (particles smaller than the wavelength of the light) is such that longer wavelengths, such as red and yellow, pass right through, but short wavelengths, such as blue, are scattered. The scattered blue light is radiated in different directions all around the sky, causing it to be the color we see.

1.2 Dispersion

This thesis focuses on one of the mechanisms that split white light into its component colors: dispersion. Specifically, chromatic dispersion is implemented, which is the phenomenon in which visible white light is separated into components of different individual wavelengths. As mentioned before, dispersion works at all the wavelengths of light, but for the purposes of this thesis just the visible spectrum is studied.

For the visible spectrum, the refractive index of most transparent materials such as air and glass decreases with increasing wavelength [Cha05]. So, a wavelength in the red range will refract at a smaller angle through glass than a wavelength in the violet range. It is for this reason we see rainbows. Air has a very small refractive index, changing the direction of light very minimally. Alternatively, diamond has a very large refractive index, changing the wave direction in a large way and creating the well-known beautiful color patterns on its surface. Figure 9 shows a diagram of a prism dispersing a ray of white light, denoted by the arrow. Figure 4 in section 1.1 shows the exact same effect – the red portion of the spectrum has refracted little compared to the violet portion.

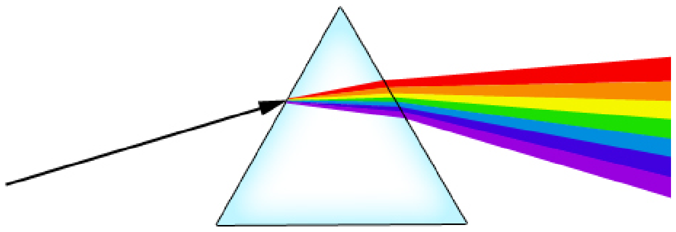


Figure 9: A diagram of a prism dispersing a ray of light. Note the angles of refraction with each color: red is affected very little, while violet's angle has changed more due to its wavelength.

In physics there are generally two types of dispersion: material dispersion and waveguide dispersion. For the purposes of this thesis only material dispersion is important; it is the type described above that causes different wavelengths to refract at different angles through a particular material. Waveguide dispersion occurs in applications such as fiber optic data transfer when the speed of a wave depends on its frequency for geometric reasons, independent of the material the wave is traveling through. From this point on, when "dispersion" is mentioned, it is material dispersion.

Dispersion can have both positive and negative effects when it comes to optics. A rainbow in the sky is an obvious example of a positive effect. However, a big issue for camera lens manufacturers for example is chromatic aberration caused by dispersion. This problem is caused by the failure of a lens to focus all of the colors (wavelengths) coming through it to the same point on film or a CCD. The result is an image that has extra unwanted colors at the boundaries between dark and light parts of a photograph. Figure 10 illustrates this problem.

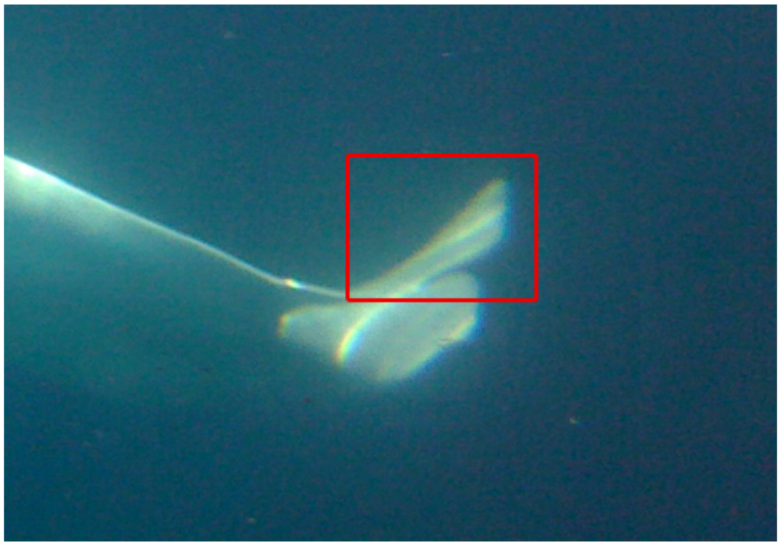


Figure 10: An example of chromatic aberration on a photograph of a whale's tail through aquarium glass. Note the red color on the left edge and the blue color on the right edge of the tail. Photo taken by author.

1.3 Spectral Rendering Challenges

Spectral rendering has remained a somewhat niche topic in computer graphics. Only within the last decade has it begun to emerge in offline renderers, for example LuxRender [Lux] and Maxwell Render [Max]. In terms of interactive or real-time applications, spectral rendering is only in the beginning stages.

Usually, during rendering, colors are represented as an RGB tuple, which essentially samples light behavior at three wavelengths (RGB) in the visible spectrum. However, spectral rendering requires samples to be taken at more points in the spectrum, which significantly increases the computational costs. The requirements of performing lighting calculations using continuous waves or multiple specific sampled wavelengths are more processor intensive than using only single photons or rays of light. This is because in the discrete world of computers, continuous functions need to be sampled at high enough levels to produce quality results.

Another performance hurtle for spectral rendering is the fact that computer displays only work with the usual tricolor RGB data. One cannot send a wavelength to a monitor and expect anything to appear on the screen. After the light transport in a scene has been calculated using waves, it has to be converted from a wavelength to an RGB value – a process that is not normally a simple one-to-one conversion if physical accuracy is the goal.

Caustics can be regarded as a specific case of dispersion, in which all the light wavelengths travel and intersect a diffuse surface in one location. With dispersion, the light wavelengths travel in different paths, possibly reaching a diffuse surface in different locations. The extra calculations required to handle dispersion beyond caustics require an extra performance cost that needs to be considered.

1.4 Goals of this Thesis

Adaptive Caustic Mapping [WN09] simulates light transport through a refractive medium in order to display caustics on diffuse surfaces. The purpose of this thesis is to extend the Adaptive Caustic Mapping algorithm so it can handle spectral dispersion. Mild approximations are made to physically accurate models to enable us to achieve real-time frame rates. The fact that there is very little published research on real-time or even interactive spectral rendering is a driving force behind this project. With the real-time synthesis of images increasing in realism and fidelity on an almost weekly basis with advances in global illumination, surface phenomena representation, and shadowing, achieving real-time frame rates for physically accurate dispersion is desirable. Challenges to overcome include accelerating all steps of spectral rendering: sampling the light wave, refracting through an object by wavelength, smoothing noise and gaps in the spectral map, and conversion from wavelength data to RGB data.

1.5 Contributions

There is one major contribution for this thesis: an interactive or real-time performance (depending on scene) method of synthesizing spectral dispersion phenomena.

1.6 Thesis Organization

This thesis is organized as follows: Chapter 2 gives a historical perspective by going over recent work in the display of caustics and spectral phenomena. Chapter 3 is an overview of how Adaptive Caustic Maps (ACMs) work, including a section on the separate – but closely related – deferred refraction algorithm used to display refractive objects. Chapter 4 is about the extensions to ACMs that add spectral dispersion effects. Chapter 5 goes into the specific implementation, and Chapter 6 is a discussion of the results. The thesis concludes and describes some areas of future research in Chapter 7. Finally, Appendix A contains shader code samples from the software written for this thesis.

Chapter 2: Related Work

This chapter describes previous research into the display of caustics and spectral rendering. First, offline rendering methods are discussed, and they are followed with interactive and real-time algorithms.

2.1 Caustics – Offline

Caustics, the focusing and de-focusing of light due to reflection and refraction, was relegated to the world of offline rendering up until around a decade ago. This light phenomenon can be seen at the bottom of a pool of water on a sunny day or on a table when bright light shines through a glass. Figure 11 is an offline rendering that displays the caustics formed when light reflects off the inside of a ring. Various offline methods that could render caustics date back to at least 1986, when Kajiya [Kaj86] mentioned their generation using path tracing in his seminal paper, "The Rendering Equation".

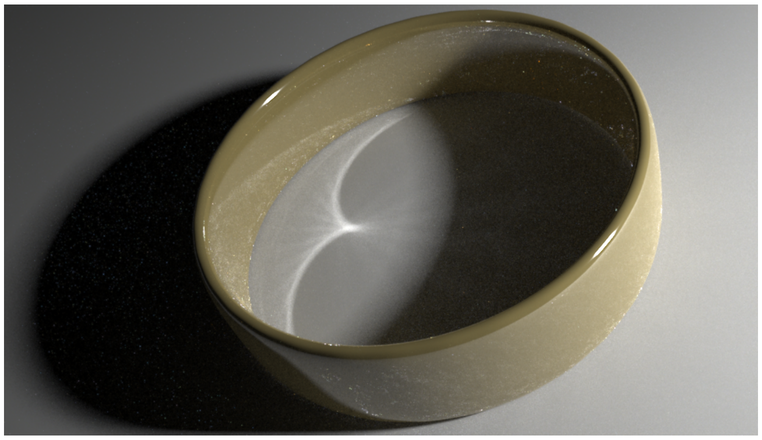


Figure 11: Caustics formed when light reflects off the inside surface of a ring. Image rendered in about 20 minutes using LuxRender.

In 1990 Peter Shirley published [Shi90] with a ray-tracing method for synthesizing entire scenes that also included the ability to display caustics. He presented an interesting three-pass method that combines conventional ray-tracing, illumination ray-tracing, and radiosity calculations. Shirley's work has a surprising similarity to this thesis – specifically in the method for finding specular objects in the scene. His algorithm first shoots "feeler" rays into the scene from the light source sparsely in all directions to find specular objects, and once they are found, illumination rays are then sent towards those objects for lighting calculations. These extra illumination rays sent towards specular objects can be used for caustic generation.

Beyond ray-tracing, and also offline, Jensen's [Jen96] work on photon mapping allowed the display of realistic caustics. Photon mapping is a two-pass algorithm: in the first pass, photons are fired into a scene and eventually rest on a diffuse surface once some termination criteria is met. The photons are stored in a photon map, which is normally a *kd*-tree data structure. In the second pass, the photons near each final image pixel are gathered and their total illumination is calculated. Photon mapping as originally proposed uses a separate special data structure called a "caustics photon map" which is separate and distinct from a normal photon map. The caustics photon map keeps track of where photons end up after refraction through a transparent medium. This is similar to the method used in this thesis, as a texture called a spectral map is utilized in which photons are splatted, or in other words the pixel in the texture where the photon lands is colored, after refraction and dispersion.

2.2 Caustics – Online

Beginning in the early 2000's researchers began to generate realistic looking caustics with specialized methods that perform at interactive or real-time frame rates. One of the first was introduced by Wand and Straßer in 2003 [WS03], who utilized the observation that the facets of a reflective object lit by a light source create light spots on diffuse surfaces that are in essence blurred images of the light source itself. They first discretize specular surfaces into sample points, and then each sample point is treated as an individual pinhole camera which projects an image of the light source onto a diffuse surface. The method is fast, but only handles light reflection, not refraction, though they mentioned it should be easy to add single-surface refraction such as from a water surface. Double-surface refraction is not easily possible with this method. This method is similar to ACMs in that only the refractive surfaces are sampled to handle the caustic calculations.

A modified version of photon mapping to display interactive caustics was published by Gunther, Wald, and Slusallek [GWS04]. Their method presents improvements to the three stages of photon mapping: photon firing, *kd*-tree construction, and the *k* nearest neighbor queries during rendering. They essentially presented a new complete framework for real-time distributed photon mapping which utilizes improvements in real-time ray tracing, photon mapping algorithms, and parallelized execution. A pitfall of their system is that it does not utilize a GPU's power and instead uses many parallel CPUs (their results were obtained with 9 to 36 CPUs working in tandem). A possibility for their choice against using GPUs could be that implementing a *kd*-tree on a GPU is not straightforward [WN09]. However, they did present an adaptive sampling strategy that accelerates the photon generation process, which is reminiscent of the adaptive sampling used in this thesis.

In 2007 Yu et al. published their paper [YLY07] on generating real-time caustics. They presented a novel algorithm that renders caustics based on a pair of caustic surfaces instead of as photons gathered on scene geometry, as most other methods (including ours) do. Caustic brightness is derived from the ray characteristic equation, the calculations of which are performed on the GPU. Their method runs in real-time and produces very good quality results, but its fairly complicated caustic generation scheme made utilizing a caustic mapping algorithm instead for a dispersion extension an easier choice.

The idea for “caustic mapping”, in which a special texture is created containing caustic data and projected onto a scene similar to shadow mapping, began with Shah et al.'s image-space technique in 2005 [SKP05]. They perform refraction calculations at each vertex of the refractive surface, then estimate the intersection point of the refracted ray with the receiver geometry, and finally estimate the intensity of the caustic at that point. This algorithm is quite fast and handles both reflective and refractive (single and double-surface) caustics. The methods presented in this thesis are based partly on Shah et al.'s research, and also perform almost completely in image-space and utilize a caustic map texture, though the generation method is quite different. In addition, the paper showed an example image using caustic mapping for a pseudo-spectral refraction effect by giving a different refractive index for each of the three tristimulus colors.

Wyman et al. introduced a similar method [WD06] also operating in image space that could display caustics at interactive rates. This approach requires two passes: one to emit particles from the light source and interact with the object, and a second to gather their contributions as seen from the eye. Wyman extended his own algorithm and introduced Hierarchical Caustic Maps [Wym08] to increase both speed and quality. This is accomplished by adjusting the number of photons utilized in the caustic calculations on the fly based on a reduced resolution version of the scene using mipmaps. The work also improved issues with over- and under-sampling due to too much photon converging or diverging after refraction. A third contribution from Wyman was Adaptive Caustic Maps [WN09], which further improve speed and quality. ACMs are discussed in detail in the next chapter, as they form the basis of this thesis.

In a method that could basically be interpreted as caustic mapping, Ming et al. [MKKLYK07] published their method for real-time display of the effects of light shining through a stained-glass window. They create a caustic map and a depth map from the light's view, blur the caustics according to distance from the stained glass (using mipmaps), and project into the scene as in a shadow map.

2.3 Spectral Rendering – Offline

Spectral rendering, specifically regarding light dispersion, began in a similar vein as the display of caustics. In 1982 Cook and Torrance published a paper [CT82] outlining their reflectance model for graphics in which they present a method for rendering that takes into account light wavelengths and spectral energy distribution. At that early date however they were only concerned with basic light transfer, not the effects discussed in this thesis.

Spencer Thomas published a paper [Spe86] specifically about spectral dispersion in 1986. He models dispersion such that when a single ray encounters a refractive object, it is split as needed into several subrays, each of which covers a small portion of the full visible spectrum. Each of these rays is possibly reflected or refracted and split again with a smaller wavelength range, and so forth until a ray intersects a diffuse surface, loses significance in the lighting calculation, or reaches a recursion depth threshold. Thomas implemented a fairly physically-accurate representation of dispersion, and as such it can be quite slow: he mentions in his paper that it is likely over 100 rays are created for a single pixel in an image.

Research beginning around the late 1980s in spectral rendering mostly focused on rendering two types of effects related to dispersion: the first being realistic and physically accurate rainbows, and the second being the display of gemstones. Thomas's paper included a very realistic image of diamonds rendered with his algorithm.

Musgrave published a paper [Mus89] in which he used a Monte-Carlo distributed ray-tracer to simulate light dispersion in raindrops for the creation of rainbows. He outlined the two largest problems that still plague spectral rendering algorithms to this day: how to sample the continuous spectrum of visible light, and how to convert wavelengths to RGB values for accurate display on a computer monitor. His solution for sampling the spectrum was to take 13 samples at specific intervals, making sure they sum to white. This thesis employs a very similar method, though seven samples are used instead: one for each of the major rainbow colors (ROYGBIV). As for converting wavelengths to RGB values, Musgrave uses the three tristimulus functions for the metameric color used to represent the color of monochromatic light of each wavelength sample. In this thesis for speed reasons a specific RGB value for each of the seven wavelengths is chosen, adding them one at a time in a shader.

In 2000 Wilkie et al. published [WTP00] a paper specifically about rendering dispersion effects using a simple ray tracer. Their spectrum sampling varied in order to show the effects of using a small number of regularly-spaced deterministic samples vs. a stochastic sampling method. They only discussed internal dispersion, that which you see on the surface of a prism, for example, as opposed to the light that refracts out of it.

Radziszewski et al. presented [RBA09] a full-spectrum technique for synthesizing images that was optimized for quasi-Monte Carlo sampling of the spectrum. This sampling method can effectively represent both smooth light distributions and lights with narrow spikes in their spectra, such as neon bulbs. Their method produced physically-accurate results for wave effects such as dispersion and scattering,

Other wavelength-dependent effects using offline algorithms have been researched as well. Smits et al. [SM92] and Hirayama et al. [HKYYM99] published work on thin film interactions. Both presented a physically-based approach to displaying thin film phenomena that takes into consideration the spectral composition of light. Dong [Don06] presented a method for combining both RGB rendering and spectral rendering in the same image, and proved it could work by implementing a multilayer thin film ray tracer. For scattering, the phenomenon that creates our blue sky, Nishita et al. published [NND96] on the display of clouds. They take into account multiple anisotropic scattering due to particles in the clouds, and also utilize sky light in addition to direct light from the sun. Stam [Sta99] and Agu [Agu01] have presented methods for displaying physically accurate diffraction, both in a ray-tracing context.

2.4 Spectral Rendering – Online

The most recent contribution in the rendering of rainbows was published in 2010, over

20 years after Musgrave. Kanamori et al. [KFYRTK10] published a similar rainbow generation paper which goes far beyond Musgrave's original research, in which they take into account such details as the radius of each raindrop, the volume (density) of raindrops in the atmosphere based on meteorological data, and scattering due to air molecules and aerosols.

The accurate display of gemstones and their dispersion properties has received much attention in graphics research. In 2004 Guy and Soler published [GS04] a paper with their algorithm using the GPU to display physically-based light transport in gemstones in real-time. They accurately display the internal reflection and refraction properties of gemstones, including effects due to the polarization of light. Their algorithm uses the concept of *facets* to represent a set of points inside a gem that corresponds to the succession of transformations a photon of light travels. The facets are grouped into a tree structure for each frame and all their contributions are accumulated using a fragment program to reach the final pixel color. While their results are fantastic in terms of “internal” light phenomena (effects inside the gemstone), this algorithm stops short of representing dispersion on external surfaces.

In 2005 Ďurikovič et al. presented a spectrally-based framework for image synthesis that runs on the GPU [ĎK05]. Their method performed at interactive frame rates and accurately displayed multilayered thin film interference effects, utilized the spectral power distribution of a light source, and included the metamerism of surfaces. They precomputed a large amount of the data required for rendering and stored it in 1D arrays in textures for quick fetching in a fragment shader.

Ihrke et al. published a paper in 2007 [IZTTMS07] on their wave-based light transport algorithm they called Eikonal Rendering. The algorithm can account for the emission, absorption, reflection, and scattering effects of light using a set of ordinary differential equations based on the Eikonal equation. While this algorithm does make use of an infrequently-utilized “wavefront model” for image formation (allowing effects otherwise more difficult in mainstream methods) and runs in real-time, it still has a few pitfalls. These include the inability to move the light source without 5-10 seconds of pre-computation time, a somewhat complex structure, and being only suitable for simulation of spatially confined refractive objects.

Possibly the most similar (in terms of final result) previous work to what is presented in this thesis is Sikachev et al.'s 2008 paper [STI08] that describes their algorithm for rendering spectral caustics on planes. It performs at interactive rates and accurately simulates light entering a refractive object, splitting into various wavelengths based on physical properties, and projecting onto a plane upon exiting the object. While their algorithm displays more realistic gems or refractive objects than what is presented in this thesis, it only projects dispersion onto planes, as opposed to arbitrary surfaces, as presented here.

As with the offline algorithms, much research has been performed on displaying spectral phenomena other than dispersion interactively or in real-time. Lindsay et al. published [LA06] in 2006 on rendering spectral diffraction effects. Using the GPU they presented a way to perform physically-based diffraction in real-time (with some precomputation of SH coefficients), preserving high dynamic range illumination. They adaptively sample the diffraction BRDF and precompute it to a Spherical Harmonic basis to preserve the full intensity of the reflected light. Iwasaki et al. presented [IMN04] a method in 2004 for rendering soap bubbles in real-time that takes into account wave-based thin film interference. They precalculated and stored the reflectivities of the thin film of the bubbles as textures in order to speed rendering.

Chapter 3: Adaptive Caustic Maps

We chose to extend Adaptive Caustic Mapping [WN09] in particular because it solved several issues inherent in other caustic mapping algorithms: notably aliasing due to insufficient sampling and excessive temporal noise due to sampling variations. ACMs use an importance-based adaptive photon sampling algorithm that increases quality while also speeding up the rendering of caustics beyond other methods of similar quality. In addition, they utilize a deferred rendering process that displays refractive objects more quickly than other methods.

3.1 Caustic Generation

Wyman introduced Adaptive Caustic Maps as a deferred shading method to improve the speed of caustic mapping. He also included an efficient deferred method for displaying refractive objects, which will be detailed in the next section.

The general structure for using ACMs to create caustic effects is similar to other caustic mapping techniques. Refractive objects are rendered from the view of the light source, and a method is used to place photons on the object. The photons are refracted through the object and splatted onto the caustic map, which is then projected onto the scene like a shadow map.

Where ACMs differ from other caustic mapping algorithms is in the photon emission and refractive object “locating” phase. Other caustic mapping algorithms normally fix the number of photons prior to emission and send them throughout the entire light view, often wasting computation time because photons that may not actually hit the refractive object are still processed.

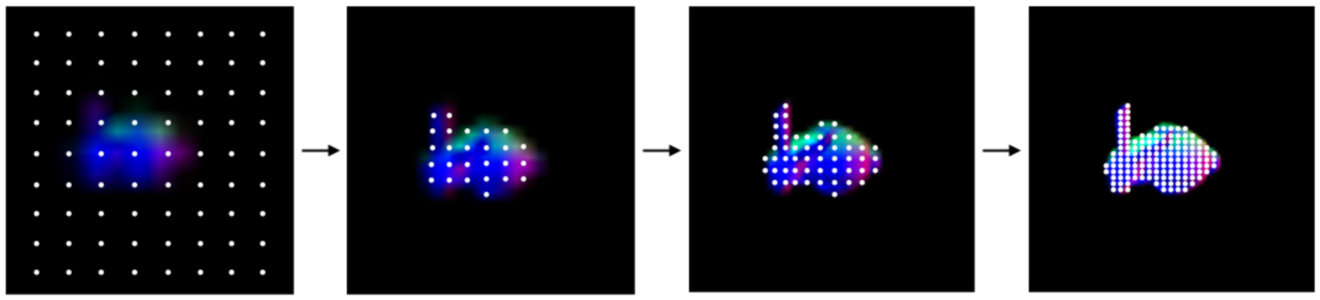


Figure 12: Photon traversal and refinement. In the first stage, very few photons are evenly spread through the scene in order to find the refractive objects. Each subsequent level refines the photons that hit the refractive object.

ACMs start with a reduced resolution view of the scene from the light using mipmaps, and only emit a few regularly-spaced photons into that image. In a loop, moving up one mipmap level at a time, each photon that actually hits a refractive object is subdivided into four new photons. Photons that do not hit a refractive object are simply discarded, never to be processed. When this is complete, the photon buffer contains a high-resolution set of points that all intersect with the surface of a refractive object. Figure 12 is an illustration of this process on the bunny model. These photons are all then refracted through the object and splatted (the texel corresponding to the photon's final location is colored) into the caustic map.

The caustic map is projected into the scene in a very similar fashion to shadow mapping. It differs from shadow mapping in that no depth test is required for checking what the light can "see", since the caustics may appear anywhere in the scene. Depending on refractive index, object shape, and light source, caustics may form both inside an object's shadow and/or in direct light. Figure 13 shows screen captures with a caustic map and the final scene with the map projected into it.

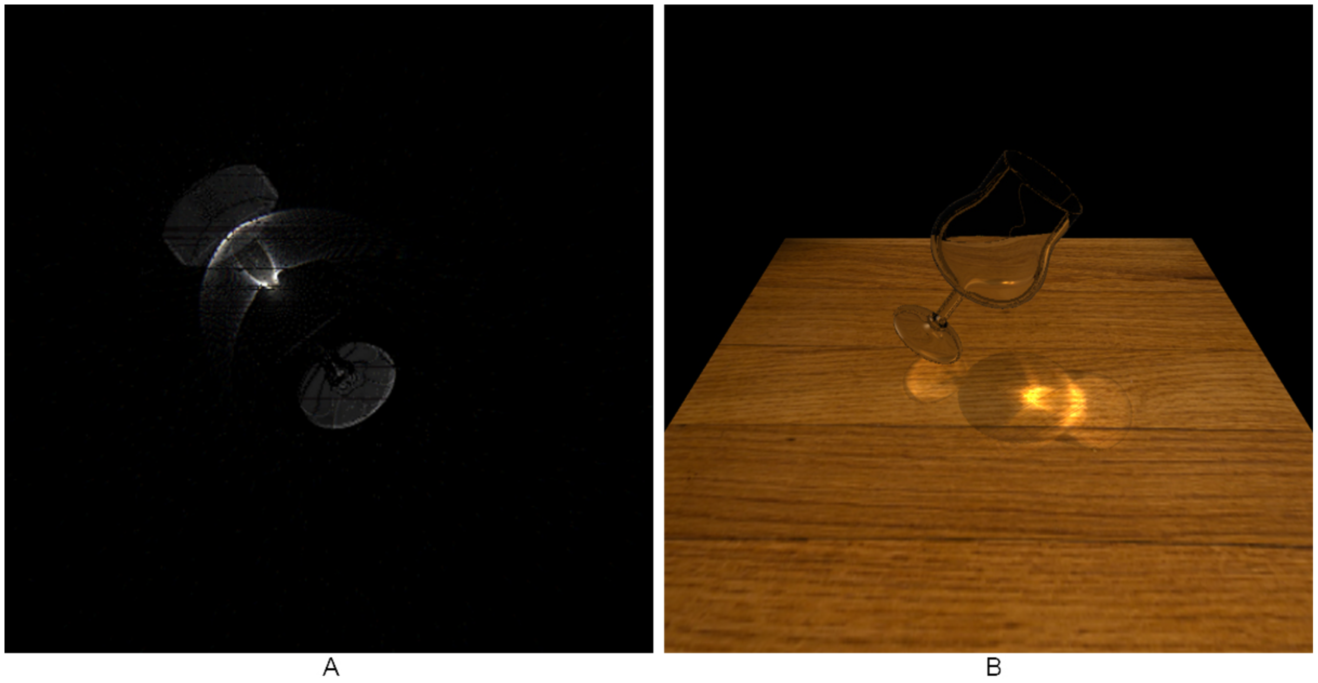


Figure 13: The caustic map, A, and final image with caustic map projected, B, for the beer glass scene. 30 fps.

3.2 Deferred Refraction

The display of the refractive objects in the scene is completed in a separate pass at the end, after the caustics have been projected onto the background geometry. It uses information already collected for caustic formation, and so does not require extra data or special textures. Input for this algorithm includes the front and back-facing normals for the refractive objects, which are held in a single texture array and gathered in one pass before caustic calculations. Depth textures of front and back-facing surfaces of the refractive objects are also needed, along with a depth texture of the background geometry.

In essence, pixels that lie on the surface of a refractive object are treated as photons, just like with the caustics calculations. Using the front normal, the photon is refracted at that front surface. The depth textures allow calculation of the distance between the front and back surfaces of the object, which are used for the photon's path to check for intersection with the back-facing surface. At the back-facing surface, the photon is refracted again, after which it is projected out to the background geometry. The point at which it hits the background geometry is then set as the color for that pixel on the refractive object.

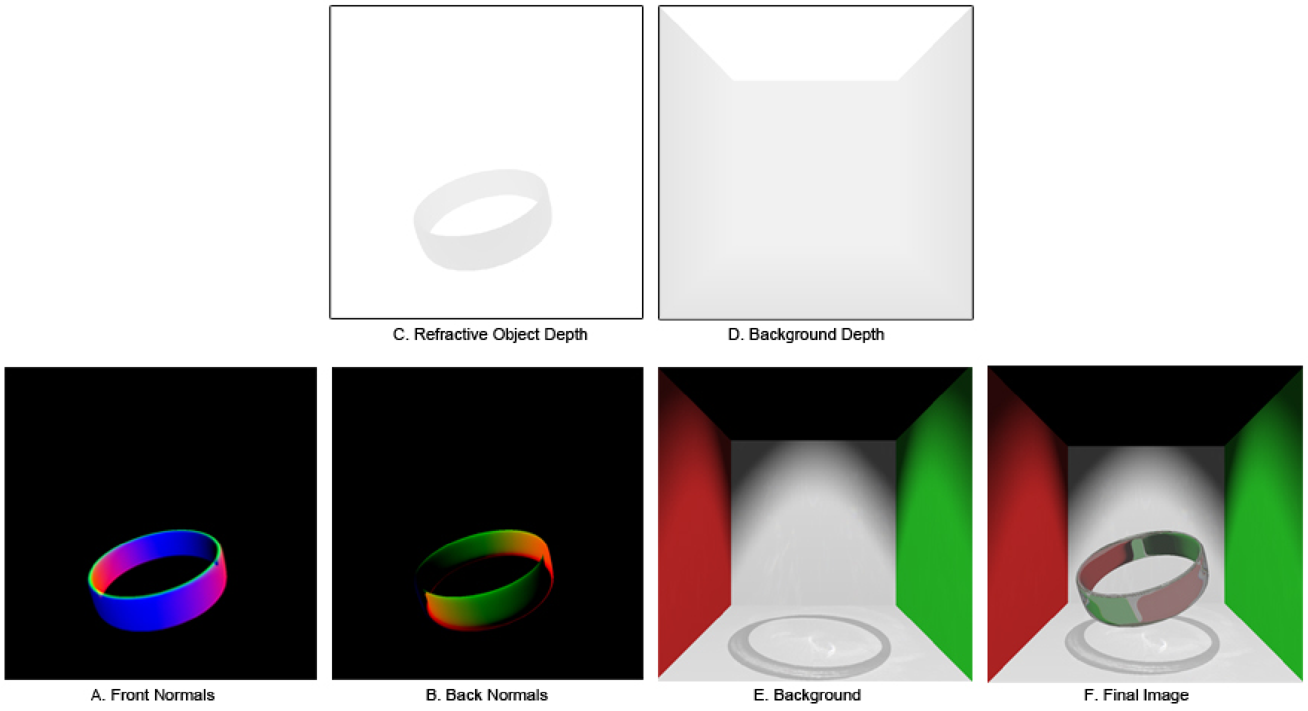


Figure 14: The image data required for deferred refraction. A and B are the front and back-facing normals, respectively, of the refractive object. C and D are the depth maps for the refractive object and background, respectively. E is the background color texture, with shadows and caustics already complete, and F is the final result.

Figure 14 shows images representing the data needed for deferred refraction. Parts A through E of Figure 14 are all flat textures: data gathered from previous render passes, encoding the data required for refraction and coloring.

Chapter 4: Spectral Dispersion

4.1 Creating Dispersion Phenomena

To extend ACMs with spectral dispersion in order to create the new spectral maps, changes were made to both the caustic generation algorithm and the deferred refraction algorithm. Specifically, the caustic generation algorithm was extended to handle what this thesis will call “external” dispersion, or that which is produced by light exiting a refractive object and landing on a diffuse surface. This is the type of dispersion seen when a prism is put into white light. The deferred refraction algorithm was extended to handle “internal” dispersion, which is the phenomena that, for example, creates the colors seen inside diamonds. Figure 15 illustrates the distinction between these two types of dispersion.

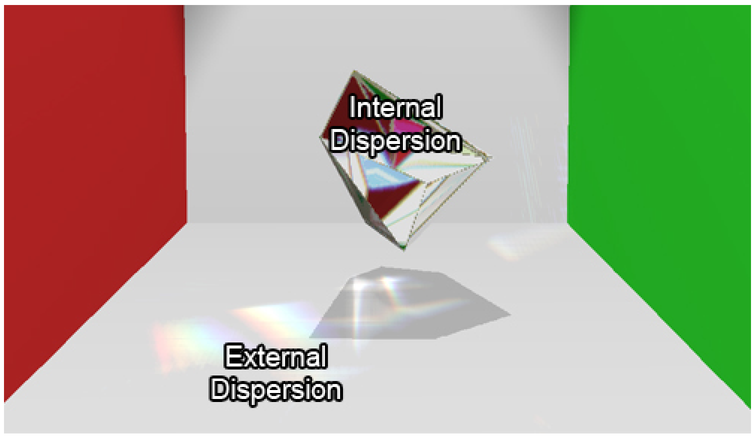


Figure 15: The distinction between "internal" dispersion, inside the refractive object, and "external" dispersion, on the diffuse surface.

The spectral dispersion algorithm begins as a choice of how many samples of the continuous visible light spectrum to utilize. As outlined above, the visible spectrum ranges from around 400 nm wavelength for violet up to around 700 nm for red. For the examples in this thesis seven samples were chosen which are evenly distributed through the spectrum. The number of chosen samples is essentially arbitrary, though with fewer than seven there tends to be too many missing colors, and with more the speed loss becomes a major problem. These issues are discussed further in the next section.

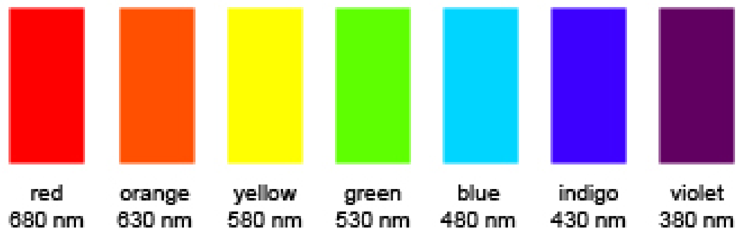


Figure 16: The seven chosen wavelength samples.

Our seven samples each correspond to a different color of the rainbow: red, orange, yellow, green, blue, indigo, and violet, and each has a specific wavelength. Figure 16 shows the wavelengths and colors chosen from the spectrum.

Refraction of light between two mediums with different refractive indices, regardless of wavelength, can be described using Snell's law [Hec01]. Snell's law is represented by the following equation along with figure 18:

$$n_1 sin\Theta_i = n_2 sin\Theta_t$$

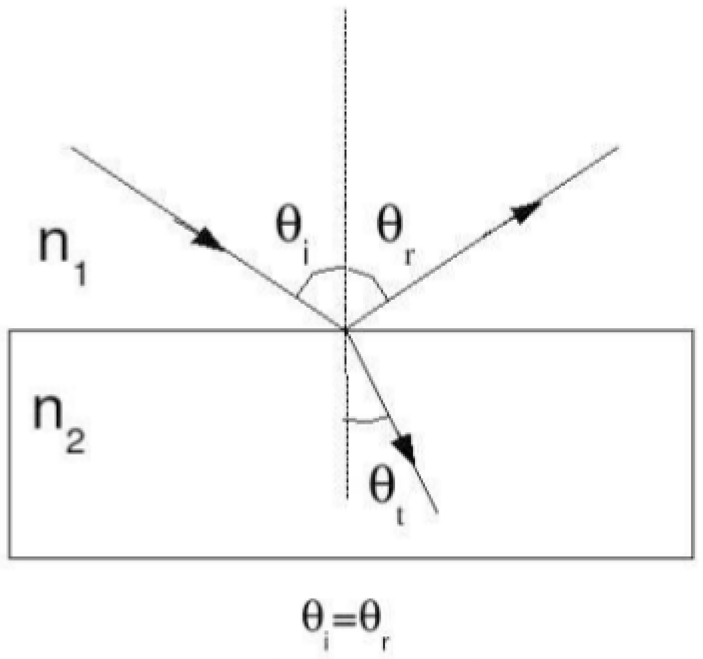


Figure 17: Refraction of light according to Snell's law. Light traveling through medium n_1 at incident angle Θ_i is refracted when entering medium n_2 at angle Θ_t. Image from [Hec01]

Where n_1 and n_2 are the refractive indices of the mediums. Figure 17 shows that light from medium n_1 at incident angle Θ_i is refracted when entering medium n_2 at angle Θ_t.

The refraction angles in Snell's law, when considering light dispersion, are wavelength dependent. These angles, taking into account wavelength, can be calculated using Cauchy's equation [Cha05] [Hec01], which describes an empirical relationship between a visible spectrum wavelength and the refractive index of a particular material:

$$n(\lambda)=A+\frac{B}{\lambda^2}+\frac{C}{\lambda^4}+...$$

Where λ is the wavelength, and A, B, C, etc. are coefficients specific to a particular material. Note it was found Cauchy's equation only works for the visible spectrum, not the entire spectrum of light. Since only the visible spectrum is required for this thesis, it is adequate.

The above is the general form of this equation, but for the purposes of this thesis it is sufficient to use the following two-term form [Mus89]:

$$n(\lambda) = A + \frac{B}{\lambda^2}$$

The A and B coefficients are based on physical measurements, and tables of them can be found in various sources such as physics books and the Internet. Table 1 shows some examples for common glass types.

Material	A	B (um)
Fused Silica	1.4580	0.00354
Borosilicate glass BK7	1.5046	0.00420
Hard crown glass K5	1.5220	0.00459
Barium crown glass BaK4	1.5690	0.00531
Barium flint glass BaF10	1.6700	0.00743

Table 1: Cauchy A and B coefficients for various types of glass. Data from [Goo10].

After the photons have all been emitted, positioned in the hierarchy on the refractive object, and ready to be refracted and splatted, the next stage of the algorithm takes place. Each photon is caught just as it is about to be refracted through the front surface of the object. The photon is split into seven separate photons, one for each wavelength, and each one is refracted according to the index of refraction generated for it from Cauchy's equation. Each of these seven photons is then refracted a second time on the back-facing surface of the object, after which its final position on the background geometry is sent to the final stage for splatting into the spectral map. Figure 18 illustrates this process, and compares it to Adaptive Caustic Mapping.

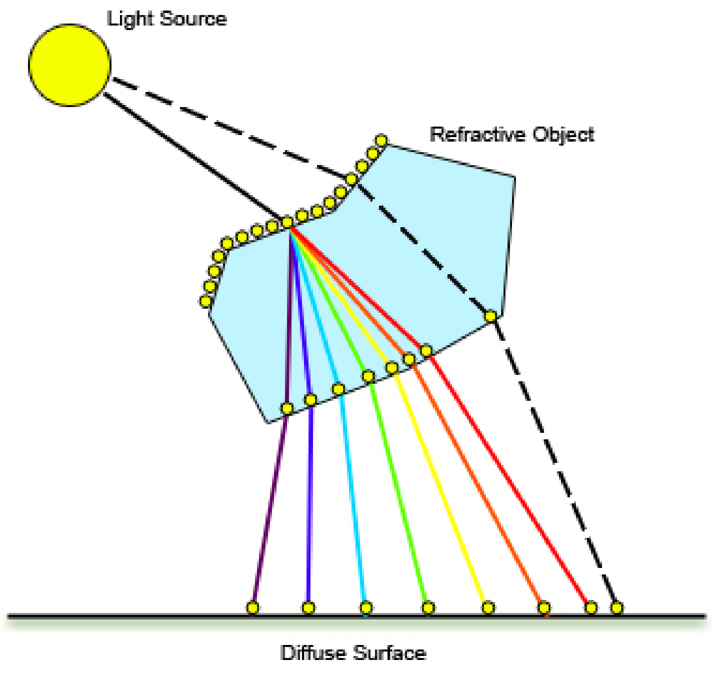


Figure 18: Refraction using seven samples, illustrating how different wavelengths are refracted at different angles in an external dispersion context. The final locations on the diffuse surface make up the spectral map. The dotted line indicates how original ACMs worked.

In Figure 18 note how each individual refracted ray's final position on the diffuse surface is in a unique location. Depending on material - specifically the refractive index of that material, and the shape of the object, all (or most) of the rays may still land in nearly the same location. In this case the colors would all be added back together, producing white. It is for this reason that the RGB values for each wavelength must be carefully calculated so that they sum to white. If they do not, the results will be incorrect.

Because only seven samples are being utilized, the distance between where each specific wavelength ray intersects the diffuse surface matters. Issues can sometimes arise in which the caustics have gaps between each color. Figure 19 illustrates this problem.

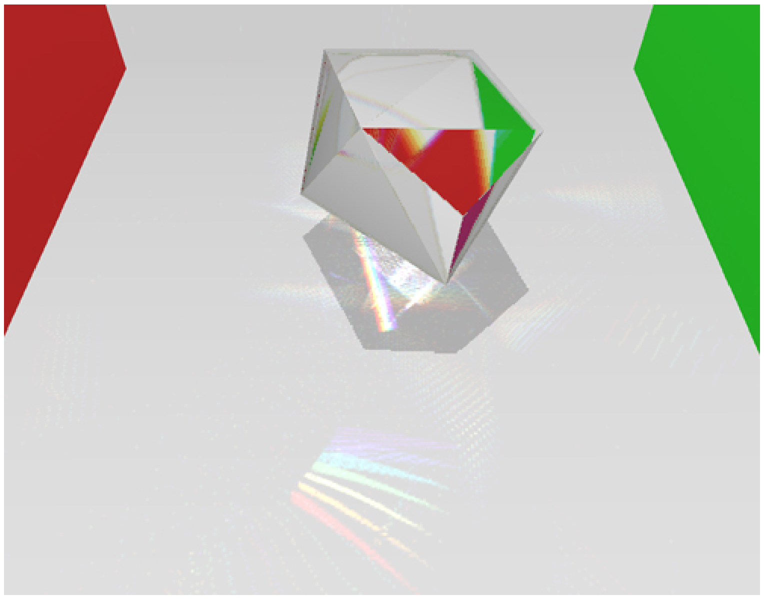


Figure 19: Problems with discontinuous caustics when using seven samples. Each sample color is clearly visible, with gaps between the colors. Note the chromatic aberration visible in the red portion of the "gem" as well.

The next section outlines and describes the novel algorithm for handling these sampling problems on the spectral caustic map.

Once the spectral map has been created and projected into the scene like a shadow map, internal dispersion is calculated in a completely separate pass at the end. No spectral map is required because we are only coloring the pixels on the surface of the refractive object. In ACMs the color of each pixel on the surface of the refractive object is calculated using a single background color texture fetch, however in ASMs seven texture fetches are performed – one for each wavelength sample. Figure 20 shows how this works. Pixel A is a representation of original ACMs, and pixel B shows how the extension works. Note that the angles of the lines are exaggerated to illustrate the process better.

Each of these texture fetches may be in a slightly different location, akin to the photons for each wavelength being splatted into a different location in the spectral map. The color of the texel chosen from the background texture is altered by the color of the wavelength that hits it, so if all wavelengths arrive at the same location, or the same color as in Figure 20 (because they add up to white), then the color is exactly the same as the background. The wavelength is converted to an RGB value here, when it is calculated based on the color of the background texture. The largest effect is seen where the background texture has a transition between light and dark colors, because the separate photons of different wavelengths may hit both the light and the dark side. This can be seen in Figure 19, where the red and gray walls intersect as viewed through the gem an orange color is present.

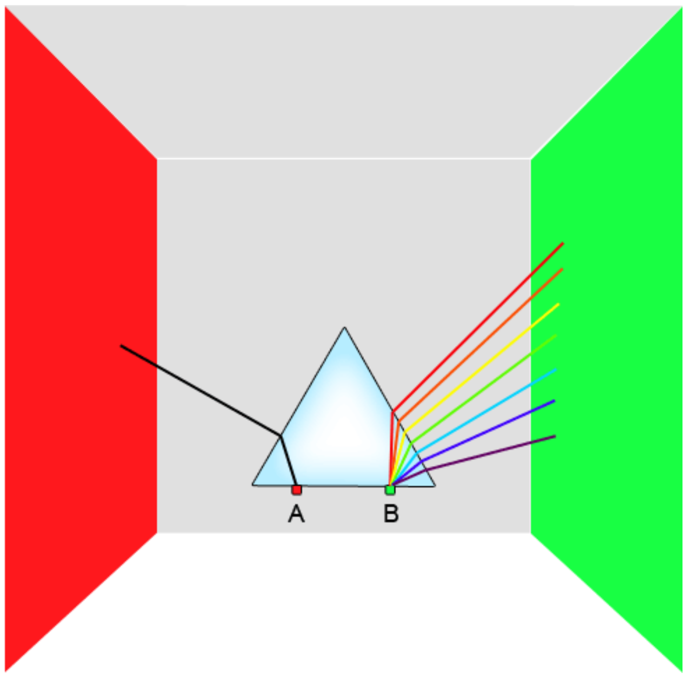


Figure 20: Deferred dispersive refraction. View is from above, looking straight down (A and B are on the front of the prism). The pixel on the surface of the prism at A shows original ACM, and the pixel at B illustrates our extension.

4.2 Filling the Gaps

One of the major issues with spectral rendering due to discrete sampling of the spectrum is gaps or empty portions in the resulting spectral map. As shown in figure 19 it can be a visually disturbing problem, breaking the impact and realism of the image. Note the absence of gaps in the rainbows directly under the refractive object: this is a problem that must be solved to work for any situation, detecting whether gaps are apparent in the caustics or not while performing a fix.

An effective solution is to increase the number of wavelength samples taken along the visible spectrum. By far the simplest solution, this indeed results in fewer gaps and holes, more colors, and greater physical accuracy. Unfortunately, it also slows everything down: like any graphics effect one is attempting to simulate, the more samples used, the more processor intensive and the slower the performance. With offline applications this is not a problem (just wait longer...), but real-time performance is being attempted. In addition, [STI08] mentioned that for their algorithm, gaps still existed even when using 20 wavelength samples. To check on this claim, the algorithm was tested with 21 samples, and figure 21 shows the result.

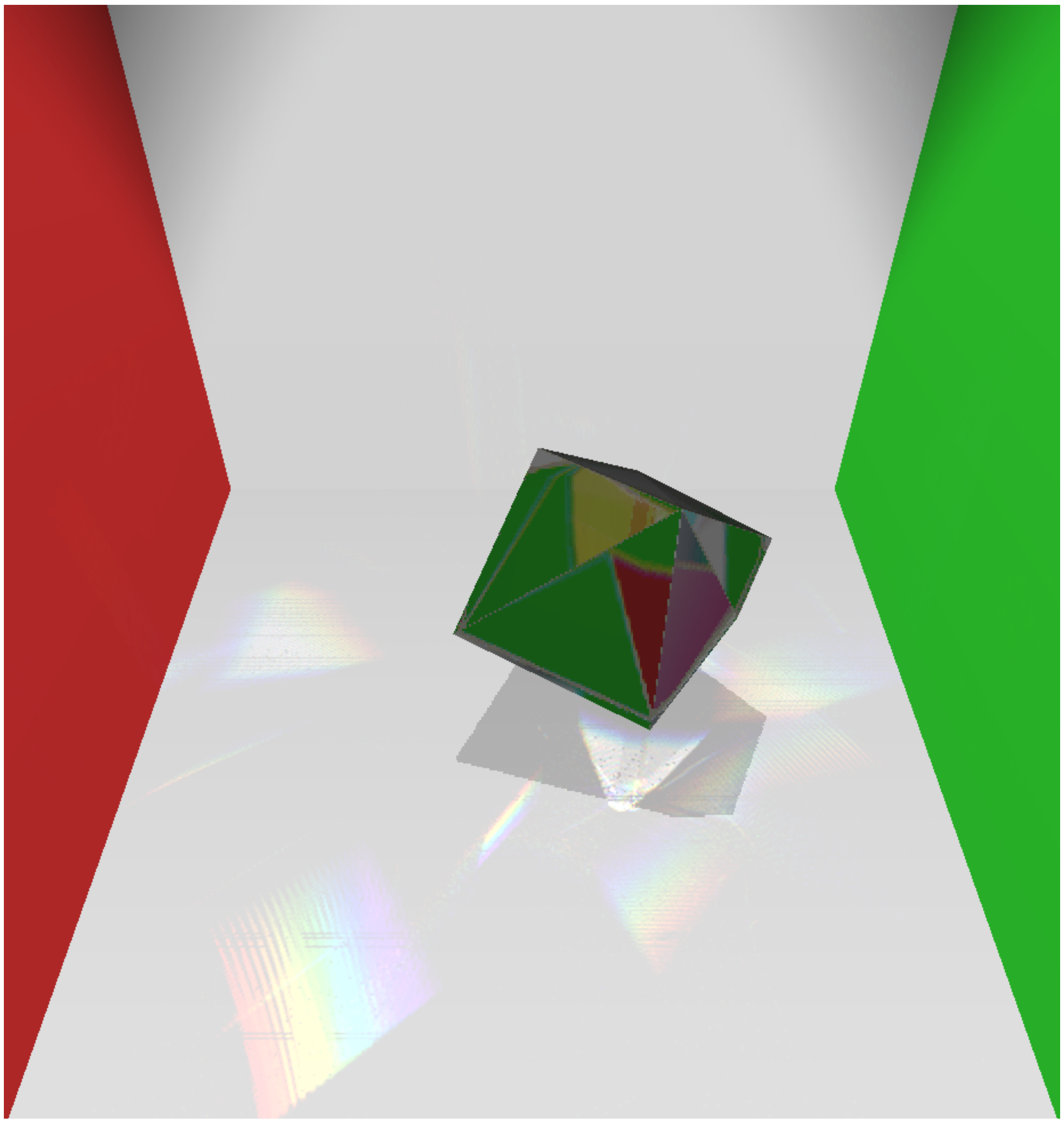


Figure 21: Adaptive Spectral Mapping using 21 evenly-distributed wavelength samples.

As can be seen in Figure 21, the worst large gaps have actually been reduced to a great degree. Where there is a large dispersion amount however, there are still some visible gaps between each color band – notably on the bottom left of the image, but they are small. However, what the image does not convey is the rather large frame rate drop when using 21 samples. The gem object in Figure 19, with seven samples, was performing at 40 frames per second, while Figure 21, with 21 samples, was performing at 10 frames per second. Using the greater number of samples did succeed in removing most large gaps, but it dropped performance from real-time to interactive. Further frame rate performance data comparing seven and 21 spectrum samples can be found in the Results section below.

Besides increasing the number of wavelength samples, various methods have been proposed. In [STI08] again Sikachev et al. proposed interpolating colors between the caustics that do exist in order to solve the color gap problem. They integrate the interpolation results for each point by performing additive blending, and use a given step size which is taken in the view space coordinates.

We propose a similar, but slightly different, method: for each pixel in the caustic map that is not already illuminated, a test is performed to check whether it is within a gap between caustic splats. If the pixel is found to be in a gap, then it is colored based on the splats near it. In essence, a step is performed one pixel at a time, horizontally and vertically from the current pixel. Along the way, if a colored pixel is found, its color is mixed with the current pixel's color. It is akin to “smearing” the colors surrounding a gap towards each other, or interpolating the colors around a gap to fill it in. This filling is accomplished completely in image space, with the only input being the spectral caustic map generated in the previous pass. Output is the spectral map texture with gaps filled in. Pseudocode for the algorithm is shown in Figure 22.

```
1. fetch spectral map texel at current pixel location;
2. if (color present) { set pixel to that color; } else {
3.  step horizontally left and right, adding color on the way;
4.  step vertically up and down, adding color on the way;
5.  mix verticalColor and horizontalColor;
6.  set current pixel to mixed color;
7. }
```

Figure 22: Pseudocode for the gap-filling algorithm.

A diagram illustrating the idea behind the gap-filling algorithm is shown in Figure 23:

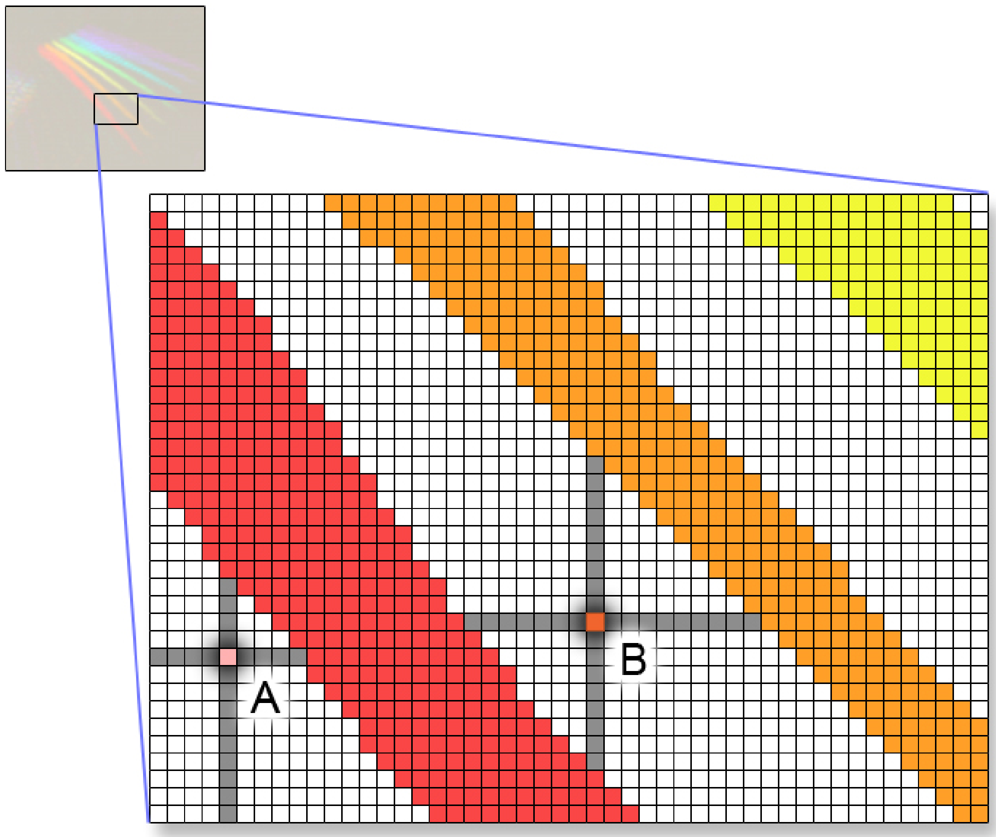


Figure 23: A diagram showing how the filling algorithm works. Each grid square is one pixel in the spectral map.

The algorithm begins with parts 1 and 2 of the pseudocode simply checking whether the pixel currently being processed already has color present, and if it does, just output that color. If no color is present, the possibility exists that the pixel is within a rainbow's color gap. Part 3 of the pseudocode is to make a set number of steps horizontally from the current pixel, one pixel at a time, to the left and right. Any colored pixels encountered on the way are added to the current pixel. Part 4 is the same process, only up and down vertically. The gray pixels in Figure 23 represent these two steps. Part 5 mixes the horizontally-found colors and the vertically-found colors, and part 6 sets the current pixel to the mixed color. In Figure 23, pixel A is set to a light red color, because its vertical and horizontal neighbors are either from the red band or from nothing at all. Pixel B is set to a mixture of the red and orange color bands due to its proximity to both colors in the spectral map. Figure 24 shows screen captures of the spectral map before and after the gap-filling procedure.

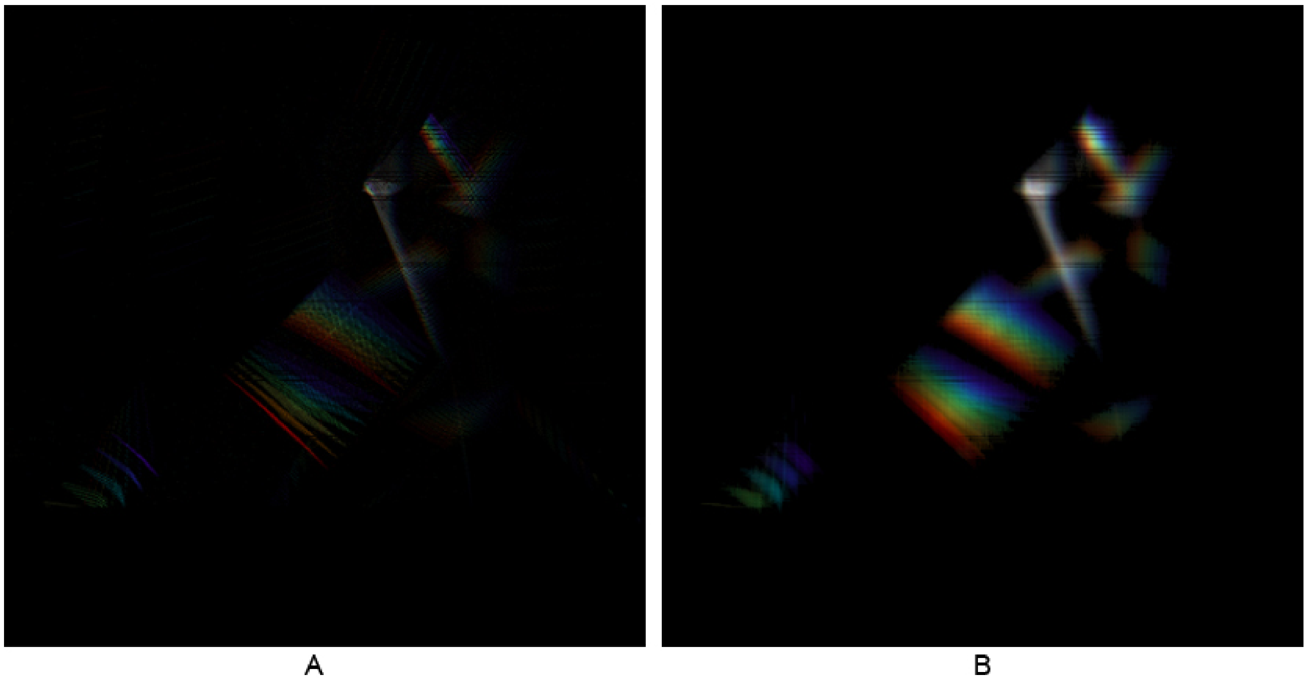


Figure 24: The spectral map texture, before (A) and after (B) filling in the gaps.

A certain efficiency is inherent when using this algorithm, because a form of blurring is essentially built-in. Since the pixels check for what is around them in order to pick a color and fill in gaps, blurring is basically given for free in the process. This helps with spots in the spectral map that may have contained single-pixel noise before the filling algorithm is performed, smoothing out inconsistencies.

The blurring can also be an issue in some cases, because there are situations where sharply defined caustics are desired. There is in fact a tradeoff here: the number of steps, or pixels to check outwards from the current pixel, matters in this case. A large step value can fill in large gaps, but also creates a more blurred edge on the spectral colors. A smaller step amount will give sharper edges, but any large gaps will not be completely filled in.

Chapter 5: Implementation

This thesis was implemented using C/C++ and OpenGL 4.2, with vertex, geometry, and fragment shaders written in GLSL. The video card utilized was a NVIDIA GeForce GTX480 in a Windows 7 environment.

The implementation began with Wyman's Adaptive Caustic Mapping algorithm [WN09], and great thanks to him for providing a code download on his website to use as a reference. His shaders were altered and extended to handle spectral dispersion – specifically to handle multiple wavelength samples instead of just single photon calculations. The photon splatting shaders were extensively modified by inserting a new geometry shader to perform the photon splitting into seven samples and to handle the refraction for each wavelength. The fragment shader was altered to handle the extra photons and convert the wavelength values to RGB.

The ACM code was also altered to make it faster, separate from the specific dispersion extensions. Instead of performing photon refinement on six mipmap levels of the refractive object texture, only three are traversed. This had two effects: almost an order of magnitude speed increase (in some scenes going from 2 fps to 12 fps), and a reduction in caustic quality. The quality decrease specifically meant dimmer caustics and more “holes”, or missing pixels, in them. Much of this was taken care of with the gap-filling shader.

After creating the spectral map and before projection onto the scene's background geometry, the novel gap-filling shader is inserted. It works directly on the spectral map itself in image-space. Figure 25 shows the entire pipeline for this project from beginning to final image.

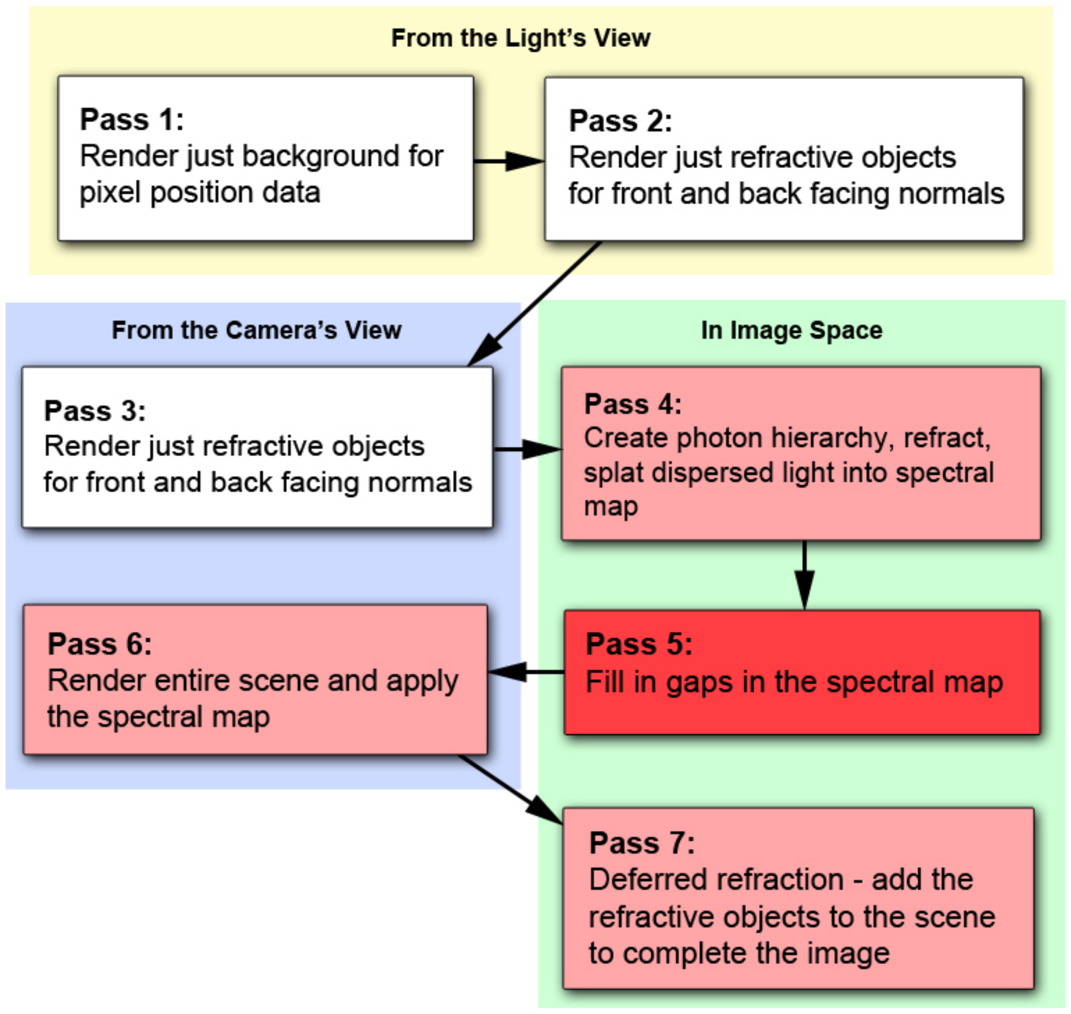

From the Light's View
Pass 1:
Render just background for pixel position data
Pass 2:
Render just refractive objects for front and back facing normals
From the Camera's View
Pass 3:
Render just refractive objects for front and back facing normals
In Image Space
Pass 4:
Create photon hierarchy, refract, splat dispersed light into spectral map
Pass 5:
Fill in gaps in the spectral map
Pass 6:
Render entire scene and apply the spectral map
Pass 7:
Deferred refraction - add the refractive objects to the scene to complete the image

Figure 25: The rendering pipeline from beginning to end. This whole pipeline is completed each frame. This diagram shows both the Adaptive Spectral Mapping algorithm and the original ACM algorithm: the red boxes are passes altered from original ACMs, and pass 5, with the dark red background, is a completely new pass.

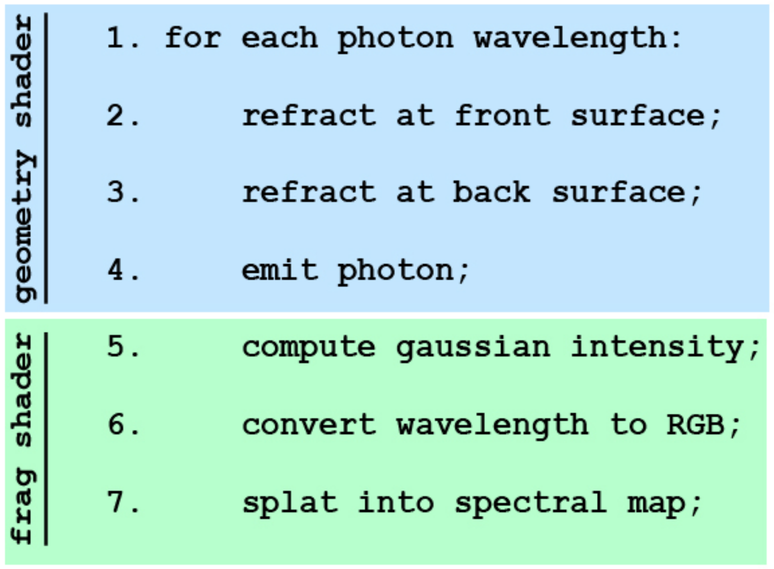
Each box in Figure 25 describes a separate render pass. The yellow, blue, and green boxes in the background show how those passes are being rendered – whether it is from the light's view, from the camera's view, or in image space (on a full-screen quad). This diagram also compares the Adaptive Spectral Mapping algorithm with the original ACMs: each white box is unaltered from the original ACM algorithm, light red boxes are altered from the original ACMs, and pass 5, the dark red box, is a completely new pass. Code samples for the altered shaders can be found in Appendix A. As can be seen in the figure, all of the actual spectral dispersion calculations described in this thesis are performed in the image space passes.

The two “light view” passes and the first “camera view” pass utilize simple shaders that only output the specific required information into textures. The 2nd and 3rd passes, where the front and back normals are written out, use texture arrays to hold the data. In this way both front-facing normals and back-facing normals can be written to the same texture in the same pass, thus increasing efficiency and only using one OpenGL texture unit. Layer 0 of the texture array holds the front-facing data, and layer 1 holds the back-facing data. This is possible due to OpenGL's geometry shader functionality allowing one to choose the layer in a texture array to write to on the fly. Incidentally, it was discovered during implementation that the specific call, gl_Layer, is flawed on ATI video cards (as of late 2011) and more often than not produces unknown or random results in the textures. Implementation in fact began with an ATI Radeon HD 6750M video card, but when this bug was discovered, the code was transferred to a computer with the aforementioned NVIDIA GTX480. As such, this thesis will currently only work on NVIDIA GPUs. Note that no specific NVIDIA video card function calls are being performed, so if ATI fixes the bug, this thesis will also then work with ATI cards.

The fourth pass, in which all the photon processing and splatting takes place, is composed of two separate but connected parts. The first part is photon traversal, described above in section 3.1, where the photons are placed on the refractive objects and refined to increase resolution where it is required. This part of the ACM algorithm was essentially left untouched and runs as Wyman and Nichols [WN09] originally wrote it.

Our spectral dispersion extensions were added to the second part of the ACM algorithm, where the photons are actually refracted through the object and splatted into the spectral map. In the original ACM code, the photon positions gathered in the traversal phase were refracted twice in the vertex shader: once through the front of the refractive object and again through the back, and then their positions were sent to the fragment shader for splatting after a simple gaussian filtering. The extension first does away with any processing in the vertex shader, instead moving all these calculations into a newly-inserted geometry shader.

This geometry shader contains most of the ACM code that was originally in the vertex shader, but it was moved here in order to support splitting the single original photon into seven separate wavelength photons and emitting them all individually. New inputs to this shader are the refractive indices for each individual wavelength, which are precomputed in the C++ code. The process is outlined in the pseudocode in figure 26.



```
1. for each photon wavelength:
2.      refract at front surface;
3.      refract at back surface;
4.      emit photon;
5.      compute gaussian intensity;
6.      convert wavelength to RGB;
7.      splat into spectral map;
```

Figure 26: Photon splat pseudocode.

Part 1 in figure 26 was added in order for the algorithm to work with all seven wavelengths instead of just one photon, as with the original ACM code. Part 2 was altered in order to use the GLSL built-in refract() function instead of the ACM author's custom refraction function for speed reasons (it's not as physically accurate, but the visual difference is quite minimal). Specifically, the difference is that the ACM author's refraction function takes care of the case where an incident ray reflects off the object's surface, whereas the built-in GLSL function does not. Part 3 still uses the ACM author's more physically accurate refraction function, but both parts 2 and 3 were edited to use the Cauchy equation-calculated refractive indices per wavelength. Part 4 is a geometry shader requirement, just there to emit the photon/vertex so the fragment shader will see each and every photon. Part 5 is identical to the ACM author's original code. In the newly-created part 6, the wavelength for that photon is converted into an RGB value. Part 7 is a simple call to gl_FragData, required for all fragment shaders.

For many of the examples in this thesis, Cauchy coefficients corresponding to fused silica glass were chosen, in which A = 1.4580 and B = 0.00354. Note that a higher B coefficient corresponds to a larger dispersion amount. Some of the examples use this feature of the Cauchy equation in order to more effectively illustrate the dispersion phenomenon, making rainbows far more likely to show up and easier to see. For these, the same A coefficient as the fused silica glass was chosen, but an extremely high B coefficient of 0.6 was utilized. Figure 27 shows the difference in dispersion between these two values of the Cauchy coefficients. For speed reasons, the algorithm pre-computes the refractive indices for each wavelength for each Cauchy coefficient value and sends that data to the shaders as an array at run-time. This allows the dispersion amount to be changed by the user in real time.

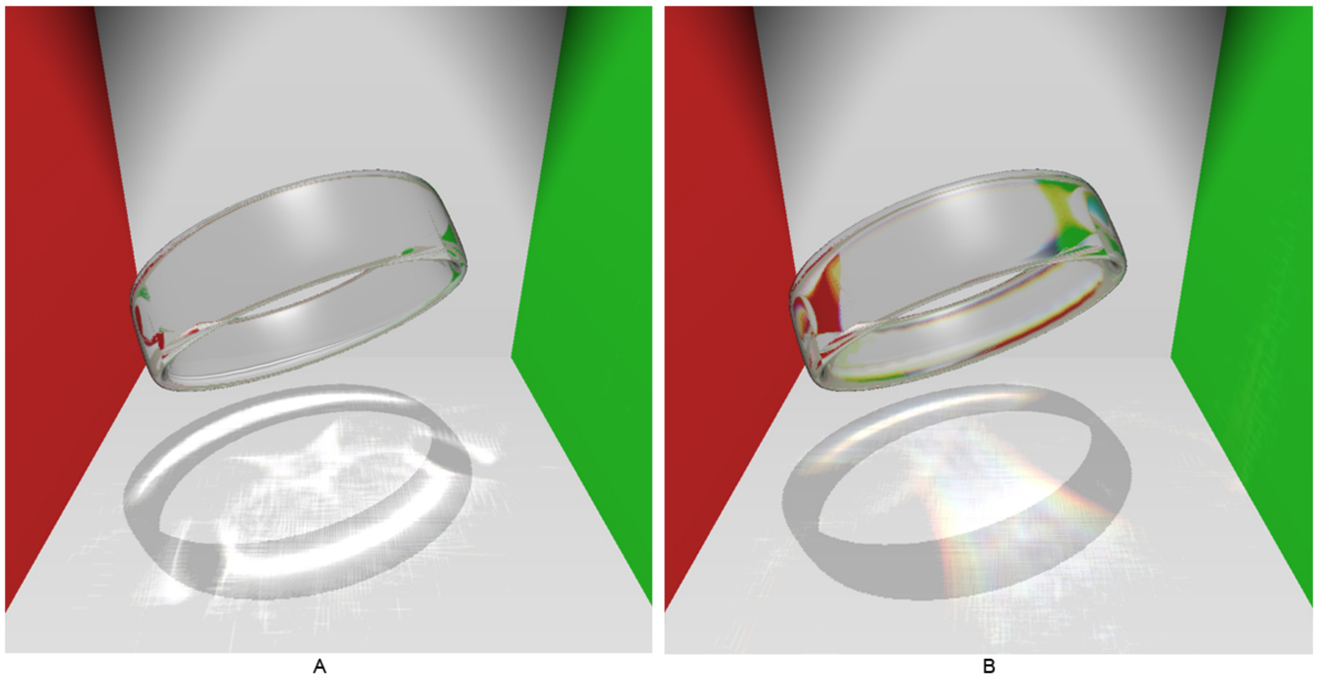


Figure 27: Comparison between dispersion that occurs with two different Cauchy B coefficients. Image A uses the real-life value for fused silica glass, 0.00354, and image B uses an extreme value of 0.6. Image B performs far more dispersion, making rainbows more likely. Dispersed colors are set to be brighter than normal for ease of comparison.

Conversion from a wavelength to an RGB value is, again for speed considerations, as simple as it gets. Because it is known exactly which wavelengths are being sent to the splat shader, a constant red, green, and blue value for each one can be set. As mentioned previously, these values must be carefully chosen to make sure they sum to white (i.e. all equal the same number) when adding all the wavelengths together as put forth in [Mus89]. If they do not sum to white, the caustics will almost certainly be too red, green, or blue, depending on the scene's circumstances. Table 2 gives the RGB values used for each wavelength. These are approximations based on using the CIE color matching functions to get the relative contributions of light from wavelength, and converting them to XYZ color space coordinates [RBA09]. From the XYZ coordinates, it is possible to get RGB values.

Wavelength (nm)	**Red**	**Green**	**Blue**
380	77	0	204
430	51	51	255
480	0	229.5	255
530	76.5	255	102
580	204	229.5	77
630	229.5	128	0
680	255	0	0
Total	893	893	893

Table 2: The RGB values chosen for each wavelength. Colors are on a 0-255 scale. Before splatting, the totals are scaled so all dispersed waves are not bright white.

In this way, a particular pixel's color in the scene is summed for each wavelength-specific photon that hits it. If all wavelengths end up on the same pixel, it will be white, as it is in physical reality. If only one photon wavelength hits a particular pixel, the pixel will only be that color.

After the spectral map is created, the next pass performs the gap-filling shader to take care of gaps and any noise or missing pixels in the spectral map. The algorithm was described in detail in section 4.2, so the specifics will not be explained here again. The shader code for this algorithm is presented in Appendix A. One note is that the number of steps to perform when searching for a color in the spectral map can be set to any value. As mentioned earlier, this is a tradeoff between blurry dispersion and better filling, or sharper dispersed colors and the possibility of gaps not completely filled in. It was found that a step size of 20 creates a good combination of gap filling and blurriness.

Following the spectral caustic map's processing, it is projected into the scene in a very similar way to a shadow map. This code is essentially unchanged from the Wyman and

Nichols [WN09] code – the only alterations involved combining a simple shadow mapping shader with their caustic projection shader so both operations are performed in the same pass.

The deferred refraction pass was also extended to handle seven separate wavelengths as opposed to single photons, as it was originally presented. In addition, a quick and simple depth check was added so that the refractive object does not appear in front of all other geometry all the time due to its deferred nature. Since no extra photons need to be created as in the caustic splatting pass, only the fragment shader's code required extension (that is, an entirely new geometry shader was not needed). No extra photons are required as in caustic calculations because the color for each pixel on the surface of the refractive object is chosen from the background geometry texture. It is the location of the fetch from the background geometry texture that is important, and seven texture fetches are performed instead of just one per pixel as with Adaptive Caustic Mapping [WN09].

The color of this pixel fetch from the background texture is altered slightly by the wavelength of the photon that hits it. Like the spectral caustic map colors, if all seven photon wavelengths hit the same pixel in the background texture, the sum will be the exact same color as the background texture. If only the “red” portion of the spectrum hits the background texel, it will be shaded more red on the refractive object image.

Gap-filling is not performed on the surface of the refractive object as it is with the spectral caustic map because gaps and missing pixels are basically non-existent. This is due to the fact that photons are not being splatted into a separate map – colors are being pulled from background geometry, which always exists (or is black if nothing is there). The only issue that may come up is non-smooth transitions between some colors, which was also observed and outlined in [WTP00]. Figure 28 shows a close-up example of this problem.

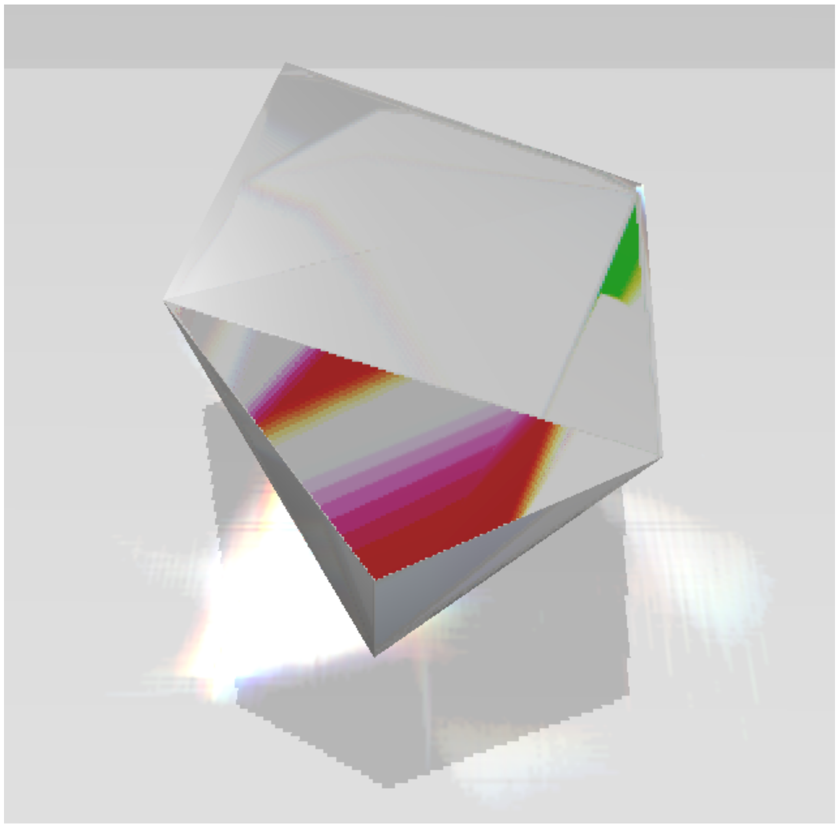


Figure 28: Example of non-smooth transitions between colors of a refracted image on the refractive object.

This issue is usually only apparent when using an extreme dispersion value (Figure 28 used the very high Cauchy B coefficient value of 0.6) and/or if the viewer gets very close to the refractive object.

Chapter 6: Results and Discussion

Table 3 contains performance data for the algorithm compared to Adaptive Caustic Mapping. Figure 29 shows all of the test scenes. Tests were performed on the Adaptive Spectral Mapping algorithm using both seven and 21 wavelength samples. As can be seen in the table, the extra photons needed for dispersion and gap-filling reduces performance in some scenes, and increasing wavelength samples severely reduced speed in all scenes. The sphere, at least with seven samples, still performs at the same speed as with ACMs, most likely due to its simplicity and the small size of its dispersion. The gem, being composed of far fewer faces than all the other objects, still performs slower than the sphere - this is most likely due to its size, which is an image-space algorithm issue discussed in the following paragraphs. The greatest difference in performance is the glass on the table, which is also due to its physical size in the light's view. Also, because the original ACM algorithm has no smoothing or blurring pass, it can display the scenes with better performance.

Object	Number of Faces	Adaptive Caustic Mapping (fps)	Adaptive Spectral Mapping (fps) with 7 samples	Adaptive Spectral Mapping (fps) with 21 samples
Sphere	5120	60	60	19
Ring	65536	27	20	9
Gem	24	60	40	10
Bunny	138902	12	10	6
Glass on Table	12137	60	16	5

Table 3: Frame rates for each test object. The sphere, ring, gem, and bunny were all refractive objects inside a Cornell box scene. The glass is on a "table" with a wood texture applied. The five test scenes can be seen in Figure 29.

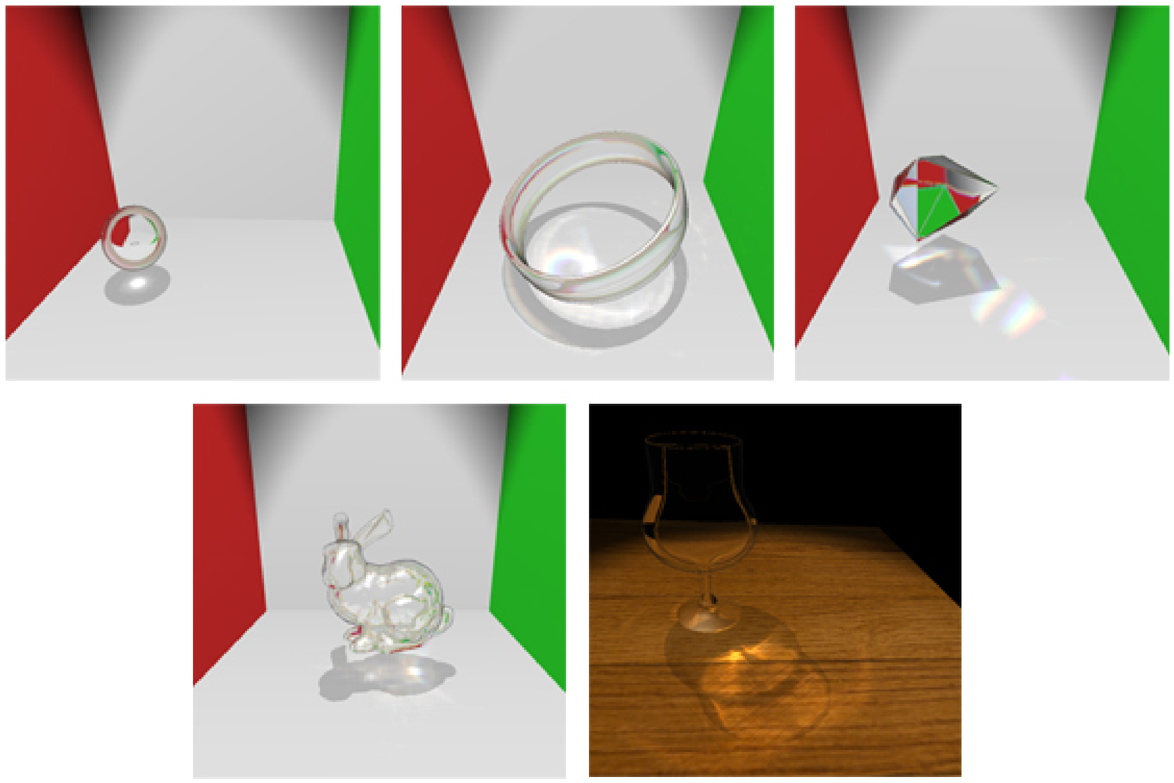


Figure 29: The five testing scenes.

Since this algorithm runs in image-space, the number of pixels covered by the refractive object from the light's view has an impact on frame rate. The closer the object is to the light, the more pixels involved in caustic calculations, and the slower the performance. In fact, the relationship between this number of pixels and frame rate is closely tied. Table 4 shows what happens as the sphere is moved closer to the light source. Figure 30 shows the sphere on the “floor” of the Cornell box along with the view from the light source for pixel comparison.

Frame rate (frames per second)	Total pixels covered by refractive object
60	2.5% (sphere on the “floor” of the Cornell box)
40	4%
30	7%
20	13%
10	33%

Table 4: Table showing relative number of pixels taken up by a refractive sphere as seen from the light source and a frame rate comparison.

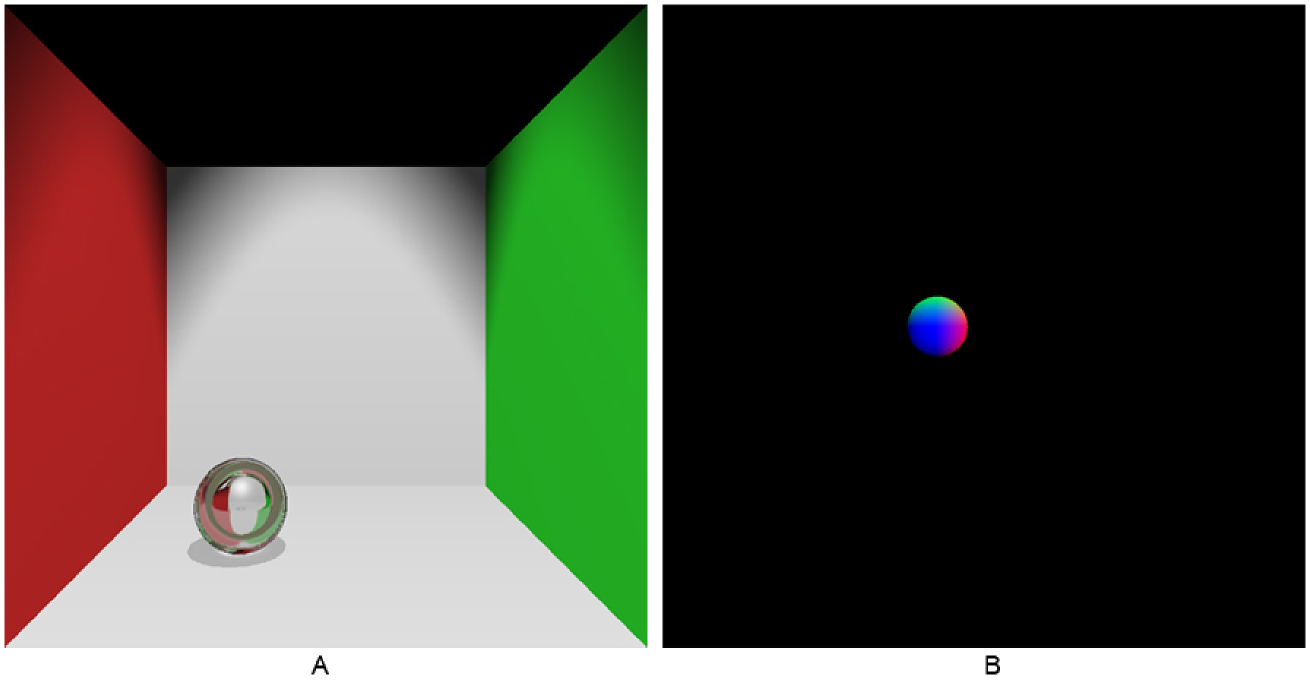


Figure 30: The sphere on the "floor" of the Cornell box, final image, A, and the view of the sphere's front-facing normals from the light, B. In this example the sphere's pixels are covering only 2.5% of the light's entire view.

The percentage of fragments covered in the light's view by the refractive object directly affects the algorithm's performance. Of course, this is closely related to the common graphics problem of quality vs. speed as well. If the refractive object is close to the light, then with this algorithm more photons will be refracted through the object, increasing the quality of the dispersion. However, as table 4 illustrates, the quality boost has severe consequences on frame rate.

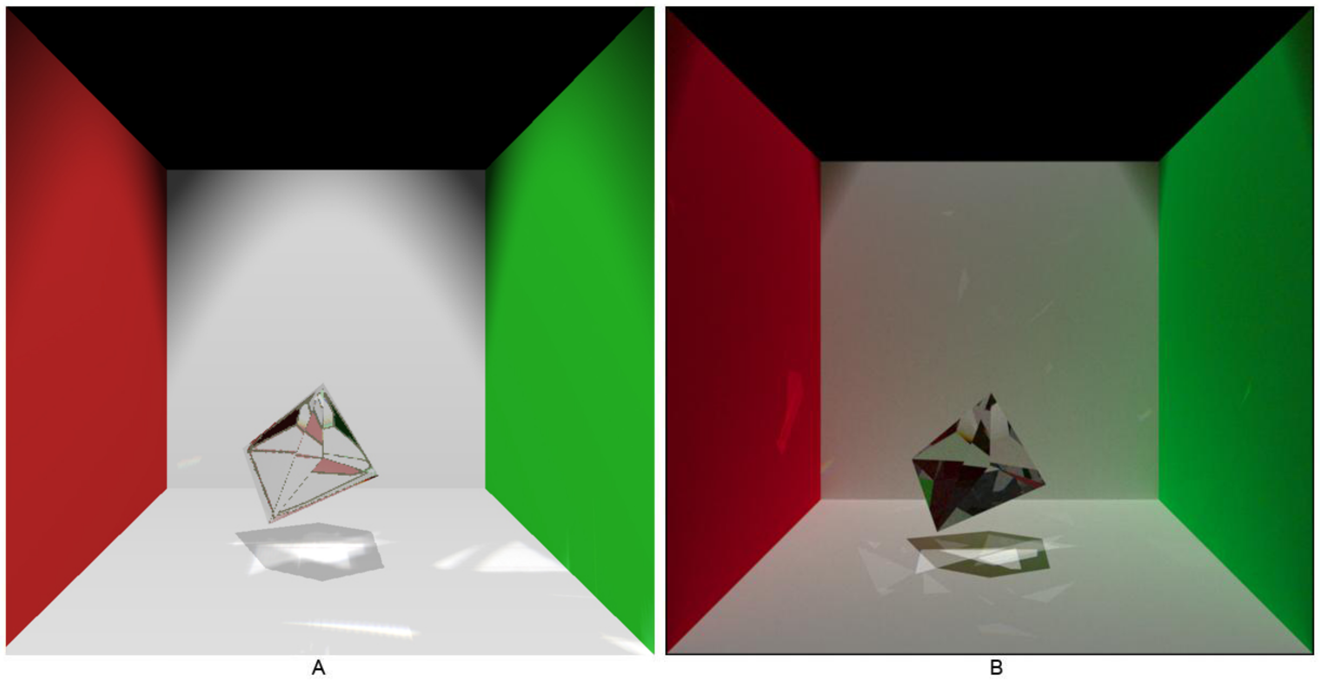


Figure 31: A comparison between the algorithm presented in this thesis, A, and a ground truth render of the same scene, B.

Figure 31 shows a comparison between a screen capture of the software and a ground truth image, which is the same scene rendered with the unbiased offline engine LuxRender [Lux]. Image A in the figure was performing at 40 frames per second, and image B took one hour to render. There are a few things to note: first is that the large caustic directly under the gem is very similar in shape, size, and color in both images. However, there are a couple discrepancies, one of which can easily be explained. The caustics on the walls in the ground truth image were created by reflective caustics, which this algorithm, and indeed Adaptive Caustic Maps as well, do not simulate. A reason for this is that reflective caustics are more difficult to simulate due to the sometimes extreme changes in a light path's direction, though this could be overcome by using a cube map for the spectral map [SKP05]. The extra caustics on the floor of the screen capture that do not appear in the ground truth are possibly there as a result of imperfect sampling of the refractive object for photon placement.

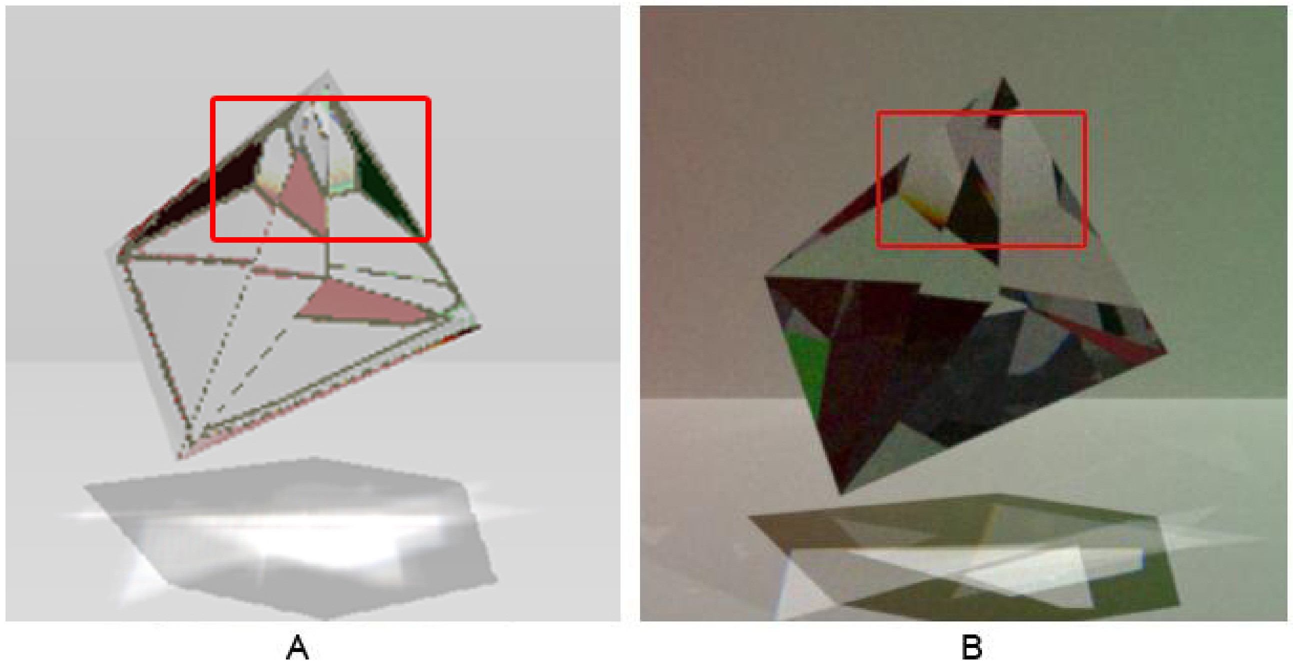


Figure 32: A close-up of the gem from Figure 31. The red box in each image highlights the spectral nature of the light calculations.

As for the gem itself, Figure 32 gives a close-up comparison. The gem rendered using the presented algorithm seems a bit pixelated – this is from the lower resolution of the refracted "image" shown in the gem. Its colors were fetched from possibly non-adjacent texels in the texture map, causing the artifacts seen in the figure. The red rectangle in each image is there to denote a similarity between the two. Inside the rectangles, on the left there is a yellowish tinge above the red color, and on the right there is a green color at the intersection of the walls. These color shifts are a result of utilizing spectral dispersion calculations, and would not be present using an algorithm that does not account for the wave nature of light. Figure 33 shows this effect in reality, as seen in the photograph of an acrylic prism.

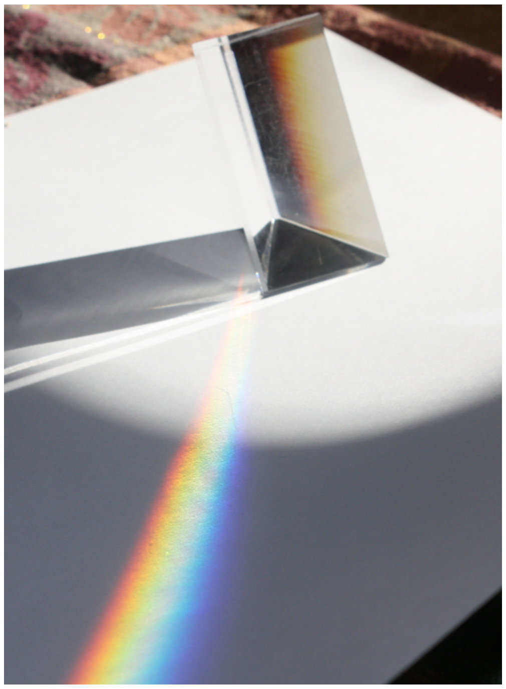


Figure 33: A photograph showing reddish color shifting on an edge between light and dark areas in the prism. Photo taken by author.

There are dissimilarities between the images as well, the most obvious being that the color of the gem to the bottom left corner of the red rectangles is different. The refraction appears to be correct however, because the lines and areas denoting refractions of the walls are quite similar.

Chapter 7: Conclusion

This thesis presented Adaptive Spectral Mapping, a spectral dispersion extension to the proposed algorithm Adaptive Caustic Mapping. Changes to both ACM itself and to the related algorithm for deferred refraction were described. The algorithm displays a plausible approximation of the dispersion phenomenon of light, and does so at interactive and real-time frame rates. The ASM algorithm is one of the first of its kind, bringing spectral rendering one step closer to being fully displayed in real-time contexts such as games.

There are some limitations to the presented algorithm, however. The gap-filling procedure creates horizontal and vertical lines in some situations due to the sampling process. This could be ameliorated with a more random sampling method, perhaps inside a certain radius around the current pixel. This would introduce a temporal cohesion issue (depending on the randomness of the sampling), but at the same time there would be fewer vertical and horizontal noise lines.

Performance is severely impacted by the number of pixels covered by a refractive object as seen from the light. This issue is common to all image-space techniques, however. A simple solution that could work in some instances would be to make sure the light is always a certain distance away from any refractive objects.

Many opportunities exist for future directions of research. The first is to extend ASMs to simulate reflective caustics, which at least one other caustics mapping algorithm [SKP05] has succeeded in accomplishing. Others include extending ASMs to display other spectral phenomena that require wavelength calculations, such as diffraction and thin-film interference. Integrating a fast volumetric caustics algorithm with ours would certainly produce beautiful images, along the lines of recent research such as [HDIGYS10]. Dispersion colors, as with shadows, become more diffuse the farther they are from the object that creates them. Any gaps between the colors also become larger the farther they are from the refractive object. A distance-aware blurring algorithm such Screen-Space Soft Shadows [GCS10] could be modified to work with the spectral maps to make them more physically accurate, and also help with very distant gaps as well.

Appendix A: Code Samples

This appendix contains GLSL shader code for the new and altered passes illustrated in Figure 25. Simple shaders that either only perform a few calls or aren't extended from the original ACM code are not presented. Most comments from the original ACM code are left in unaltered to aid understanding.

The following is the GLSL geometry shader for photon splatting into the spectral map: the second half of pass 4 in the pipeline in Figure 25. The first half of pass 4 - the creation of the photon hierarchy - is essentially unchanged from Adaptive Caustic Mapping. This code is also much the same as original ACMs (unchanged from Chris Wyman's code, including some comments), but it was in the vertex shader instead, and only output one photon instead of seven, as this code does. Original ACM code and extensions are commented where necessary.

```
// This file takes in some basic info from the vertex shader
// (basically each individual "photon" that has hit the refractive
// object) and splits it into seven new photons, each representing a
// specific wavelength. They are refracted and each photon is
// emitted, after which the fragment shader splats them into the
// spectral map.

#version 120
#extension GL_EXT_geometry_shader4 : enable
#extension GL_EXT_gpu_shader4 : enable

uniform float tanLightFovy2;

// local1 contains data for the light's near and far clipping
// distances and local2 contains index of refraction data from the
// c++.
uniform vec4 local1, local2;

// front and back facing normals of the refractive geometry
```

```
uniform sampler2DArray geomNorm;
// refractive geometry depths per pixel
uniform sampler2DArray geomDepth;
// background geometry depths
uniform sampler2D otherObjsDepth;

// spotlight texture
uniform sampler2D spotLight;

const int numWavelengths = 7;

// Since the ACM refraction function requires four separate values of
// the index of refraction, we must hold each of these for each
// wavelength, all pre-calculated in the c++
uniform float waveIndex1[numWavelengths];
uniform float waveIndex2[numWavelengths];
uniform float waveIndex3[numWavelengths];
uniform float waveIndex4[numWavelengths];

// (function from original ACMs)
// This takes an eye-space (actually light-space since
// our "eye" is at the light here) position and converts
// it into a image-space (u,v) position. This simply
// applies the GL projection matrix and homogeneous
// divide.
vec2 ProjectToTexCoord( vec4 eyeSpacePos )
{
     vec4 projLoc = gl_ProjectionMatrix * eyeSpacePos;
     return ( 0.5*(projLoc.xy / projLoc.w) + 0.5 );
}

// ACM author's more accurate refraction function
vec4 refraction( vec3 incident, vec3 normal, float ni_nt, float
               ni_nt_sqr )
{
     vec4 returnVal;
     float tmp = 1.0;
     float IdotN = dot( -incident, normal );
     float cosSqr = 1.0 - ni_nt_sqr*(1.0 - IdotN*IdotN);
     return ( cosSqr <= 0.0 ?
          vec4( normalize(reflect( incident, normal )), -1.0 ) :
          vec4( normalize( ni_nt * incident + (ni_nt* IdotN -
          sqrt( cosSqr )) * normal ), 1.0 ) );
}

// (function from original ACMs)
// Takes a screen coordinate and turns it into a 3D vector pointing
// in the correct direction (in eye-space)
vec3 DirectionFromScreenCoord( vec2 texPos )
{
     vec3 dir = vec3( tanLightFovy2 * ( 2.0*texPos - 1.0 ), -1.0 );
     return normalize( dir );
}

void main( void )
{
     // the following is unchanged from original ACMs, up to the for
     // loop going through numWavelengths.
     vec4 vertCoord = gl_TexCoordIn[0][0];
     vec3 coord = vec3( vertCoord.xy, 0.0 );
     float outside = 0.0, noBackNorm = 0.0;
     vec2 Dist;

     // Get the front facing surface normal based upon the screen-
     // space vertex position.
     vec4 tmp = texture2DArray( geomNorm, coord );

     // Get front facing refractor position
     vec4 P_1 = vec4( tmp.w*DirectionFromScreenCoord( coord.xy ),
          1.0);

     // Check if this pixel has refractive materials or not.
     outside = dot(tmp.xyz, tmp.xyz) < 0.5 ? 1.0 : 0.0;

     // Compute normalized V and N_1 values.
     vec3 N_1 = normalize( tmp.xyz ); // Surface Normal
     vec3 V = normalize( P_1.xyz ); // View direction

     // Find the distance to front & back surface, first as
     //normalized [0..1] values, than unprojected
     Dist.x = texture2DArray( geomDepth, vec3(coord.xy,1) ).z;
     Dist.y = texture2DArray( geomDepth, vec3(coord.xy,0) ).z;
     Dist = local1.x / (Dist * local1.y - local1.z );

     // Distance between front & back surfaces
     float d_V = Dist.y - Dist.x;
     vec4 projectedPhoton = vec4(0.0);
     bool invalidPhoton = false;

     // everything from here down needs to be done once per
     // wavelength since each wavelength sample has a different index
     // of refraction.
```

```
for (int i = 0; i < numWavelengths; i++) {
     // the following is unchanged from original ACMs (including
     // comments) except for the front-surface refraction and
     // the output of seven photons (and a few other small
     // changes).
     // find the refraction direction
     // 1.0003 is the refractive index of air
     vec3 T_1 = refract(V, N_1, 1.0003/waveIndex3[i]);

     // Right now, we're using a hacked hack to avoid requiring
     // d_N
     float d_tilde = d_V;

     // Compute approximate exitant location & surface normal
     vec4 P_2_tilde = vec4( T_1 * d_tilde + P_1.xyz, 1.0);
     vec3 N_2 = texture2DArray( geomNorm,
          vec3( ProjectToTexCoord( P_2_tilde ), 1.0) ).xyz;

     float dotN2 = dot( N_2.xyz, N_2.xyz );

     // What happens if we lie in a black-texel? Means no
     // normal! Conceptually, this means we pass thru "side" of
     // object. Use norm perpindicular to view
     if ( dotN2 == 0.0 ) {
          N_2 = normalize(vec3( T_1.x, T_1.y, 0.0 ) );
     }

     // Refract at the second surface
     vec4 T_2 = refraction( T_1, -N_2, waveIndex3[i],
                 waveIndex4[i] );

     invalidPhoton = T_2.w < 0.0 || outside > 0.5;
     T_2.w = 0.0;

     // Scale the vector so that it's got a unit-length z-
     // component
     vec4 tmpT2 = T_2 / -T_2.z;

     // Compute the texture locations of ctrPlusT2 and
     // refractToNear.
     float index, minDist = 1000.0, deltaDist = 1000.0;
     for (index = 0.0; index < 2.0; index += 1.0)
     {
          float texel = texture2D( otherObjsDepth,
               ProjectToTexCoord( P_2_tilde + tmpT2 *
               index ) ).x;
```

```
                float distA = -(local1.x / (texel * local1.y -
                     local1.w)) + P_2_tilde.z;

                if ( abs(distA-index) < deltaDist )
                {
                     deltaDist = abs(distA-index);
                     minDist = index;
                }
           }

           // Do our final iteration to home in on the final photon
           // position.
           // Original ACMs used 10 here, but actually using 5 seems
           // to work just as well (very small visual difference), and
           // it's faster
           for (float index = 0.0; index < 5.0; index += 1.0)
           {
                float texel1 = texture2D( otherObjsDepth,
                ProjectToTexCoord( P_2_tilde + minDist * tmpT2 ) ).x;

                minDist = -(local1.x / (texel1 * local1.y - local1.w))
                          + P_2_tilde.z;
           }

           // OK, find the projected photon position in the caustic
           // map.
           vec4 photonPosition = P_2_tilde + minDist * tmpT2;
           projectedPhoton = gl_ProjectionMatrix *
                                vec4( photonPosition.xyz, 1.0 );

           projectedPhoton /= projectedPhoton.w;

           gl_Position = projectedPhoton;
           gl_TexCoord[2] = projectedPhoton;
           gl_TexCoord[1] = vec4(i, 0.0, 0.0, 0.0);

           // spotlight color
           gl_TexCoord[3] = gl_TexCoordIn[0][3];

           EmitVertex();
           EndPrimitive();
      }
}
```

The following is the GLSL fragment shader for photon splatting, performed for each vertex emitted from geometry shader above. Like the previous shader, this is mostly unchanged from the original ACMs, including comments, except for the wavelength color calculations at the end.

```
#version 120
#extension GL_EXT_gpu_shader4 : enable

// splatResolutionModifier is an intensity modifier that results
// from changing various resolution parameters using the user
// interface, scene file, or simply traversal through the hierarchy.
// In order to keep the correct splat intensity, we need to know
// what level of subdivision has been applied to this photon.
uniform float splatResolutionModifier;

// Since our gl_FragCoord is in image space, we need to know how
// big this image is if we plan to use the gl_FragCoord to do
// useful things independent of resolution.
uniform float renderBufRes;

void main( void )
{
     // We'll fix our Gaussian splat size at just under 3 pixels. Set
     // Gaussian distribution parameters.
     float splatSize = 2.5;
     float sizeSqr = splatSize*splatSize;

     float isInsideGaussian = 0.0;

     // We need to compute how far this fragment is from the center
     // of the splat. We could do this using point sprites, but our
     // experience is the final framerate is significantly faster
     // this way. You may find differently.
     vec2 fragLocation = gl_FragCoord.xy;

     float red = 0.0, green = 0.0, blue = 0.0;
     vec4 finalColor = vec4(0.0);

     // position of the vertex output from the geometry shader
     vec4 wavePos = gl_TexCoord[2];

     vec2 pointLocation = (wavePos.xy * 0.5 + 0.5) * renderBufRes;
```

```
// Gaussian from Graphics Gems I, "Convenient anti-aliasing
// filters that minimize bumpy sampling"
float alpha = 0.918;
float beta_x2 = 3.906; // == beta*2 == 1.953 * 2;
float denom = 0.858152111; // == 1 - exp(-beta);
float distSqrToSplatCtr = dot(fragLocation - pointLocation,
          fragLocation - pointLocation);

float expResults = exp( -beta_x2*distSqrToSplatCtr/sizeSqr );

// Are we even inside the Gaussian?
isInsideGaussian = ( distSqrToSplatCtr/sizeSqr < 0.25 ? 1.0 :
                    0.0 );

// Make sure the Gaussian intensity is properly normalized.
float normalizeFactor = 10.5 * sizeSqr / 25.0;

// Compute the Gaussian intensity
expResults = alpha + alpha*((expResults-1.0)/denom);

// In original ACMs, this was calculated just before splatting,
// but in ASMs we do it here in order to apply it to each
// wavelength.
float adjustment = splatResolutionModifier * isInsideGaussian *
                    expResults / normalizeFactor;

// RGB values hardcoded in.
// seven sample version, each index in gl_TexCoord[1].x
// represents a certain wavelength.
if (gl_TexCoord[1].x == 0) {
     // violet
     red = adjustment * 0.3;
     blue = adjustment * 0.8;
}
if (gl_TexCoord[1].x == 1) {
     // indigo
     red = adjustment * 0.2;
     green = adjustment * 0.2;
     blue = adjustment * 1.0;
}
if (gl_TexCoord[1].x == 2) {
     // blue
     green = adjustment * 0.9;
     blue = adjustment * 1.0;
}
if (gl_TexCoord[1].x == 3) {
```

```
          // green
          red = adjustment * 0.3;
          green = adjustment * 1.0;
          blue = adjustment * 0.4;
     }
     if (gl_TexCoord[1].x == 4) {
          // yellow
          red = adjustment * 0.8;
          green = adjustment * 0.9;
          blue = adjustment * 0.3;
     }
     if (gl_TexCoord[1].x == 5) {
          // orange
          red = adjustment * 0.9;
          green = adjustment * 0.5;
     }
     if (gl_TexCoord[1].x == 6) {
          // red
          red = adjustment * 1.0;
     }

     finalColor = vec4(red, green, blue, 1.0);

     // finally, multiply by the spotlight color and output
     gl_FragData[0] = vec4(gl_TexCoord[3].rgb * finalColor.rgb, 1.0);
}
```

The following is the GLSL fragment shader for the gap-filling procedure, which is pass 5 in Figure 25. This shader is completely original, made for the ASM algorithm.

```
#extension GL_EXT_gpu_shader4 : enable
// the spectral map
uniform sampler2D spectralMap;

// use this shader?
uniform int useSmear;

// number of steps to go out from the current pixel
int steps = 20;

// boost (or diminish) the brightness of the filled-in pixels
float brightness = 0.2;
```

```
vec4 checkVertical(ivec2 origPos) {
     vec4 newColorUp = vec4(0.0);
     vec4 newColorDown = vec4(0.0);
     // check each pixel up from the current pixel
     for (int vUp = 1; vUp < steps; vUp++) {
          newColorUp += texelFetch2DOffset(spectralMap, origPos, 0,
          ivec2(0, vUp));
     }

     // check each pixel down from the current pixel
     for (int vDown = -1; vDown > -steps; vDown--) {
          newColorDown += texelFetch2DOffset(spectralMap, origPos, 0,
               ivec2(0, vDown));
     }

     // if the color found is bright enough, mix it in
     if ((newColorUp.r > 0.1 || newColorUp.g > 0.1 || newColorUp.b >
     0.1) && (newColorDown.r > 0.1 || newColorDown.g > 0.1 ||
     newColorDown.b > 0.1)) {
          return vec4(mix(newColorUp, newColorDown, 0.5).rgb, 1.0) *
               brightness;
     }

     return vec4(0.0);
}

vec4 checkHorizontal(ivec2 origPos) {
     vec4 newColorLeft = vec4(0.0);
     vec4 newColorRight = vec4(0.0);

     // check pixels to the right of the current pixel
     for (int vRight = 1; vRight < steps; vRight++) {
          newColorRight += texelFetch2DOffset(spectralMap, origPos,
          0, ivec2(vRight, 0));
     }

     // check pixels to the left of the current pixel
     for (int vLeft = -1; vLeft > -steps; vLeft--) {
          newColorLeft += texelFetch2DOffset(spectralMap, origPos, 0,
               ivec2(vLeft, 0));
     }

     // mix in any bright enough pixels
     if ((newColorRight.r > 0.1 || newColorRight.g > 0.1 ||
     newColorRight.b > 0.1) && (newColorLeft.r > 0.1 ||
     newColorLeft.g > 0.1 || newColorLeft.b > 0.1)) {

          return vec4(mix(newColorRight, newColorLeft, 0.5).rgb, 1.0)
               * brightness;
     }

     return vec4(0.0);
}

void main() {
     // this pixel's screen location
     ivec2 pos = ivec2(gl_FragCoord.xy);

     // color in the caustic map
     vec4 causticColor = texelFetch2D(spectralMap, pos, 0);

     if ((causticColor.r < 0.3 && causticColor.g < 0.3 &&
     causticColor.b < 0.3) && useSmear == 1) {
          // little or no color, so this pixel is possibly in a gap
          // between caustic colors
          vec4 vertColor = checkVertical(pos);
          vec4 horizColor = checkHorizontal(pos);

          gl_FragColor = vec4(mix(vertColor, horizColor, 0.5).rgb,
               1.0);
     } else {
          // just pop out caustic color
          gl_FragColor = causticColor;
     }
}
```

The following is the vertex shader used to project both the shadow map and the spectral map into the scene, which is pass 6 in figure 25. This code is a combination of standard shadow mapping, original programming, and some ACM code used for spectral map projection.

```
uniform mat4 MctoLightMatrix;
uniform vec3 LightPosition;
// ambient and diffuse scale factors
const float As = 1.0 / 1.5;
const float Ds = 1.0 / 3.0;
```

```
void main() {
     vec3 normal = gl_Normal; // world space normal
     vec4 ecPosition = gl_ModelViewMatrix * gl_Vertex;
     vec3 ecPosition3 = (vec3(ecPosition)) / ecPosition.w;
     vec3 VP = LightPosition - ecPosition3;
     VP = normalize(VP);

     float diffuse = max(0.0, dot(normal, VP));
     float scale = min(1.0, As + diffuse * Ds);

     vec3 eyeNorm = gl_NormalMatrix * gl_Normal;

     gl_TexCoord[0] = gl_MultiTexCoord0;
     gl_TexCoord[5] = ecPosition;
     gl_TexCoord[6].xyz = eyeNorm;

     // for shadow map coordinates
     vec4 texCoord = MCtoLightMatrix * gl_Vertex;
     vec4 ShadowCoord = texCoord;
     ShadowCoord.z -= 0.005;
     gl_TexCoord[2] = ShadowCoord;

     vec4 color = vec4(scale * gl_Color.rgb, gl_Color.a);
     gl_TexCoord[1] = color;

     gl_TexCoord[3] = gl_Vertex;
     gl_Position = ftransform();
}
```

The following is the fragment shader that goes along with the above vertex shader. It is also a combination of standard shadow mapping, original code, and a small amount of ACM code used for spectral map projection in particular.

```
#extension GL_EXT_gpu_shader4 : enable
uniform sampler2D causticMap;
uniform sampler2D spotLight;
uniform sampler2DShadow shadowMap;
uniform sampler2DShadow causticDepth;

uniform vec4 sceneAmbient;

// textures for different scenes if required
```

```
uniform sampler2D tex1;
uniform int useTexMaps;

void main() {
     vec4 color = gl_TexCoord[1];
     color.a = 1.0;
     vec4 ShadowCoord = gl_TexCoord[2];

     // shadow map coordinates
     vec4 smapCoord = ShadowCoord;
     smapCoord /= smapCoord.w;
     smapCoord.z = smapCoord.z <= 0.0 ? 0.0 : smapCoord.z;

     // spotlight contribution, and whoa there hold it back a touch
     vec4 spotContrib = texture2D(spotLight, smapCoord.xy) * 0.7;

     // the shadows and light
     float shadeFactor = shadow2DProj(shadowMap, ShadowCoord).z;
     shadeFactor = shadeFactor * 0.25 + 0.75;

     // this makes anything not in the light's view dark, otherwise
     // the spotlight texture shows up duplicate where the light
     // can't see.
     vec4 lightContribution = (all( equal(smapCoord.xyz,
               clamp(smapCoord.xyz, vec3(0), vec3(1)) ) ) ?
               vec4(shadeFactor * color.rgb, 1.0) * spotContrib :
               vec4(0.0));

     // and this makes the dark parts show up as just dark instead of
     // totally black
     lightContribution += (color * sceneAmbient);

     // spectral map projection, as in original ACMs
     float lightIntensity = 1.0;
     vec3 toLight = normalize( gl_LightSource[0].position.xyz -
          gl_TexCoord[5].xyz );
     vec3 norm = normalize( gl_TexCoord[6].xyz );

     float NdotL = lightIntensity * max( 0.0, dot( norm, toLight ) );
     vec4 causticContribution = texture2D(causticMap, smapCoord.xy) *
               NdotL;

     // final color, combining everything
     if (useTexMaps == 1) {
          // texture for something in the scene if required
          vec4 texContribution = vec4(1.0);
          texContribution = texture2D(tex1, gl_TexCoord[0].st);
          gl_FragData[0] = (lightContribution + causticContribution *
               3.0) * texContribution;
     } else {
          gl_FragData[0] = lightContribution + causticContribution *
               3.0;
     }

     // pixel positions
     gl_FragData[1] = gl_TexCoord[3];

     // pixel normals
     gl_FragData[2] = vec4(norm, 1.0);
}
```

The following is the deferred refraction GLSL fragment shader. It contains code mostly from the original ACM deferred refraction algorithm (including comments), with extensions to work with seven texture fetches instead of just one plus the wavelength to color conversion code.

```
#extension GL_EXT_gpu_shader4 : enable

// local1 contains data for the light's near and far clipping
// distances and local2 contains index of refraction data from the
// c++.
uniform vec4 local1, local2;

// background geometry depths
uniform sampler2DArray otherObjsEyeDepth;

// background geometry colors
uniform sampler2DArray otherObjsEye;

// spotlight texture
uniform sampler2D spotLight;

// geometry front and back facing normals and depths
uniform sampler2DArray geomNorm;
uniform sampler2DArray geomDepth;

uniform mat4 gluOrtho, shadowMatrix;
uniform float renderBufRes, tanEyeFovy2;
uniform vec4 glassColor;
uniform vec4 sceneAmbient;

// the light's position
uniform vec3 lightPosition;

// number of wavelength samples
const int numWavelengths = 7;

// Since the ACM refraction function requires four separate values of
// the index of refraction, must hold each of these for each
// wavelength, calculated in the c++
uniform float waveIndex1[numWavelengths];
uniform float waveIndex2[numWavelengths];
uniform float waveIndex3[numWavelengths];
uniform float waveIndex4[numWavelengths];

// (original ACM function)
// Get the spotlight color based upon an eye-space position
vec4 SpotLightColor( vec4 eyeSpaceCoord )
{
     vec4 smapCoord = shadowMatrix * eyeSpaceCoord;
     smapCoord /= smapCoord.w;
     return
     ( all(equal(smapCoord.xyz,clamp(smapCoord.xyz,vec3(0),vec3(1))))
          ? texture2D( spotLight, smapCoord.xy, 0.0 ):
          vec4( 0.25 ) );
}

// (original ACM function)
// Take the eye-space position and project it into a 2D image
// coordinate
vec2 ProjectToTexCoord( vec4 eyeSpacePos )
{
     vec4 projLoc = gl_ProjectionMatrix * eyeSpacePos;
     return ( 0.5*(projLoc.xy / projLoc.w) + 0.5 );
}

// (original ACM function)
// A simple refraction shader similar to the built in GLSL one (only
// this one is real)
vec4 refraction( vec3 incident, vec3 normal, float ni_nt, float
     ni_nt_sqr )
{
     vec4 returnVal;
     float tmp = 1.0;
     float IdotN = dot( -incident, normal );
     float cosSqr = 1.0 - ni_nt_sqr*(1.0 - IdotN*IdotN);
     return ( cosSqr <= 0.0 ?
     vec4( normalize(reflect( incident, normal )), -1.0 ) :
     vec4( normalize( ni_nt * incident + (ni_nt* IdotN - sqrt( cosSqr
          )) * normal ), 1.0 ) );
}

// (original ACM function)
// Approximate the fresnel coefficients using Schlick's approximation
vec2 fresnelApprox( float cosAng )
{
     float oneMinus = 1.0-cosAng;
     float approx = 0.05 +
          0.95*(oneMinus*oneMinus*oneMinus*oneMinus*oneMinus);
     return vec2( approx, 1.0-approx );
}

// (original ACM function)
// Take a 2D image-space screen position (in [0..1]) and turn it into
// an eye-space viewing direction.
vec3 DirectionFromScreenCoord( vec2 texPos )
{
     vec3 dir = vec3( tanEyeFovy2 * ( 2.0*texPos - 1.0 ), -1.0 );
     return normalize( dir );
}

void main( void )
{
     vec3 coord = vec3( gl_TexCoord[0].xy, 0.0 );
     float outside = 0.0, noBackNorm=0.0;
     vec2 Dist, fresnel;
     vec4 reflectedColor = vec4(0.0);

     // Find our surface normal on the front refractive surface.
     // If there's no refractive surface there, this shader is
     // easy -- we're done.
     vec4 tmp = texture2DArray( geomNorm, coord );

     // ASMs: changed from original ACMs here.
     // if no refractive surface, put the background pixel there.
     // it's looking at a texture that only contains the refractive
     // object - so if it doesn't see it, just throw in the
     // background texture (no need to do extra work)
     if (tmp.a == 0.0) {
          gl_FragColor = texture2DArray(otherObjsEye,
               vec3(gl_TexCoord[0].xy, 0.0));

     } else if (texture2DArray(otherObjsEyeDepth, coord).z <
          texture2DArray(geomDepth, vec3(gl_TexCoord[0].xy, 0.0)).z)
     {
          // (new in ASMs)
          // if the background is closer than the object, show the
          // background, otherwise object will always be in front of
          // everything
          gl_FragColor = texture2DArray(otherObjsEye,
               vec3(gl_TexCoord[0].xy, 0.0));
     } else {

          // Get front facing position and normal
          vec4 P_1 =
               vec4( tmp.w*DirectionFromScreenCoord( coord.xy ),
                    1.0 );

          // (this simplified for speed in ASMs)
          // Check if this pixel has refractive materials or not.
          // for ASMs every pixel that gets to this point is
          // refractive.
          outside = 1.0;

          // Compute normalized V and N_1 values.
          vec3 N_1 = normalize( tmp.xyz ); // Surface Normal
          vec3 V = normalize( P_1.xyz ); // View direction
          float NdotV = dot( -V, N_1 );

          // Compute a direction for the light (to use for a Phong
          // reflected component)
          vec3 toLight = normalize( lightPosition - P_1.xyz );
          vec3 halfVec = normalize( toLight - V );
          float NdotH = max( 0.0, dot( N_1, halfVec ) );
          vec4 reflectedLightColor = pow( NdotH, 50.0 ) *
               SpotLightColor( P_1 );

          // Find the reflective (.x) and refractive (.y) fresnel
          // coefficients
          fresnel = fresnelApprox( NdotV );

          // Find the distance to front & back surface, first as
          // normalized [0..1] values, than unprojected
          Dist.y = length( P_1.xyz );
          Dist.x = texture2DArray( geomDepth, vec3(coord.xy,1) ).z;
          Dist.x = 2.0 * local1.x / (Dist.x * local1.y - local1.z );
```

```
// Distance between front & back surfaces
float d_V = -Dist.y - Dist.x;

// compute the reflection direction and the reflection
// color we do a matrix multiply to account for (potential)
// user rotation of the environment
tmp = vec4( reflect( V, N_1 ), 0.0 );
reflectedColor = fresnel.x * (20.0 * reflectedLightColor +
     2.0 * sceneAmbient);

// ASMs:
// here, where the refraction takes place, is our time to
// split and run through all the wavelength samples
vec4 finalColor = vec4(0.0, 0.0, 0.0, 1.0);

// ASMs - go through each wavelength, one at a time.
for (int i = 0; i < numWavelengths; i++) {
     // find the refraction direction
     // glsl refract() may be less accurate, but close
     // enough to not notice and it's faster
     // 1.0003 is the refractive index of air
     vec3 T_1 = refract(V, N_1, 1.0003 / waveIndex3[i]);

     // (these comments and mention of the SIGGRAPH paper
     // are from the original ACM code)
     // Our approximation of d_tilde is different than that
     // given in the SIGGRAPH paper. It seems there's not
     // usually any need to interpolate between d_V and d_N
     // -- instead d_V alone works surprisingly well just
     // about as often. Plus, this approach requires no
     // precomputation.
     float d_tilde = d_V;

     // Compute approximate exitant location & surface
     // normal
     vec4 P_2_tilde = vec4( T_1 * d_tilde + P_1.xyz, 1.0);
     vec3 N_2 = texture2DArray( geomNorm,
          vec3( ProjectToTexCoord( P_2_tilde ), 1.0) ).xyz;
     float dotN2 = dot( N_2.xyz, N_2.xyz );

     // What happens if we lie in a black-texel? Means no
     // normal! (d_tilde is too big...)
     if ( dotN2 == 0.0 )
     {
          // Conceptually, we pass thru the "side" of the
          // object (not front/back)
          // Use a 'normal' perpendicular to view direction
          // (but generally along same direction as our
          // refracted direction T_1)
          tmp = vec4( T_1.x, T_1.y , 0.0, dot(T_1.xy,
               T_1.xy) );
          N_2 = tmp.xyz / tmp.w;
     }

     // Refract at the second surface
     vec4 T_2 = refraction( T_1, -N_2, waveIndex3[i],
          waveIndex4[i] );

     bool TIR = T_2.w < 0.0;
     T_2.w = 0.0;

     // Scale the vector so that it's got a unit-length z-
     // component
     vec4 tmpT2 = T_2 / -T_2.z;

     // Compute the texture locations of ctrPlusT2 and
     // refractToNear.
     float index, minDist = 1000.0, deltaDist = 1000.0;

     for (index = 0.0; index < 2.0; index += 1.0)
     {
          float texel = texture2DArray( otherObjsEyeDepth,
               vec3(ProjectToTexCoord( P_2_tilde + tmpT2 *
               index ), 0.0) ).x;

          float distA = -(local1.x / (texel * local1.y -
               local1.w)) + P_2_tilde.z;

          if ( abs(distA-index) < deltaDist )
          {
               deltaDist = abs(distA-index);
               minDist = index;
          }
     }

     float distOld = minDist;
     for (float index = 0.0; index < 10.0; index += 1.0)
     {
          float texel1 = texture2DArray( otherObjsEyeDepth,
               vec3(ProjectToTexCoord( P_2_tilde + distOld
               * tmpT2 ), 0.0) ).x;
```

```
                distOld = -(local1.x / (texel1 * local1.y -
                local1.w)) + P_2_tilde.z;
            }

            vec4 transmitColor = vec4( exp(-d_V * glassColor.a) *
                glassColor.rgb, 1.0);

            vec4 refractedColor = transmitColor *
                texture2DArray( otherObjsEye,
                vec3(ProjectToTexCoord( P_2_tilde + distOld *
                tmpT2 ), 0.0) );

            // ASMs - convert wavelength to color.
            if (i == 0) {
                refractedColor *= vec4(0.3, 0.0, 0.8, 1.0);
            }
            if (i == 1) {
                refractedColor *= vec4(0.2, 0.2, 1.0, 1.0);
            }
            if (i == 2) {
                refractedColor *= vec4(0.0, 0.9, 1.0, 1.0);
            }
            if (i == 3) {
                refractedColor *= vec4(0.3, 1.0, 0.4, 1.0);
            }
            if (i == 4) {
                refractedColor *= vec4(0.8, 0.9, 0.3, 1.0);
            }
            if (i == 5) {
                refractedColor *= vec4(0.9, 0.5, 0.0, 1.0);
            }
            if (i == 6) {
                refractedColor *= vec4(1.0, 0.0, 0.0, 1.0);
            }

            // scale so it's not too bright or too dark
            // (why 0.26? It's a matter of each of the above R, G,
            // and B values adding up to 3.5)
            refractedColor *= 0.26;

            finalColor += vec4(refractedColor.xyz, 1.0) *
                fresnel.y;

        } // end for loop for numWavelengths
```

```
          gl_FragColor = vec4(reflectedColor.xyz + finalColor.xyz,
               1.0);

     } // end if statement for background color
}
```

Bibliography

[Agu01] Emmanuel Agu. Diffraction Shading Models in Computer Graphics. *Electronic Doctoral Dissertations for UMass Amherst*, 2001.

[Cha05] Germain Chartier. Introduction to Optics, Springer, 2005.

[Cro10] Benjamin Crowell. Light and Matter Vol. 5, Optics, Fullerton, 2010.

[CT82] Robert Cook and Kenneth Torrance. A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics, Vol. 1, No. 1, p. 7-24*, 1982.

[DD08] Eustace Dereniak and Teresa Dereniak. Geometrical and Trigonometric Optics, Cambridge University Press, 2008.

[Don06] Weiming Dong. Rendering Optical Effects Based on Spectra Representation in Complex Scenes. *Advances in Computer Graphics*, 2006.

[GCS10] Jesus Gumbau, Miguel Chover, and Mateu Sbert. Screen Space Soft Shadows. *GPU Pro, p. 477-490*, 2010.

[GMN94] Jay Gondek, Gary Meyer, and Jonathan Newman. Wavelength Dependent Reflectance Functions. *SIGGRAPH '94 Proceedings of the 21st annual conference on Computer graphics and interactive techniques, p. 213-220*, 1994.

[Goo10] Jan W. Gooch. Encyclopedic Dictionary of Polymers, Volume 1, Springer, 2010.

[GS04] Stephane Guy and Cyril Soler. Graphics Gems Revisited: Fast and Physically-Based Rendering of Gemstones. *ACM Transactions on Graphics, Vol. 23, No. 3, p. 231-238*, 2004.

[GWS04] Johannes Gunther, Ingo Wald, and Philipp Slusallek. Realtime Caustics Using Distributed Photon Mapping. *Eurographics Symposium on Rendering (2004), p. 111-122*, 2004.

[HDIGYS10] Wei Hu, Zhao Dong, Ivo Ihrke, Thorsten Grosch, Guodong Yuan, and Hans-Peter Seidel. Interactive Volume Caustics in Single-Scattering Media. *I3D '10 Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, p. 109-117*, 2010.

[Hec01] Eugene Hecht. Optics, 4th Edition, Addison Wesley, 2001.

[HKYYM99] H. Hirayama, K. Kaneda, H. Yamashita, Y. Yamaji, and Y. Monden. Visualization of Optical Phenomena Caused by Multilayer Films with Complex Refractive Indices. *Computer Graphics* *and Applications, p. 128-137*, 1999.

[IMN04] Kei Iwasaki, Keichi Matsuzawa, and Tomoyuki Nishita. Real-time Rendering of Soap Bubbles Taking into Account Light Interference. *Proceedings of Computer Graphics International, p. 344-348*, 2004.

[IZTTMS07] Ivo Ihrke, Gernot Ziegler, Art Tevs, Christan Theobalt, Marcus Magnor, Hans-Peter Seidel. Eikonal Rendering: Efficient Light Transport in Refractive Objects. *ACM Transactions on Graphics, Vol. 26, No. 3, p. 59-1 - 59-9*, 2007.

[Jen96] Henrik Wann Jensen. Global Illumination using Photon Maps. *Rendering Techniques '96, p. 21-30*, 1996.

[Kaj86] James Kajiya. The Rendering Equation. *SIGGRAPH '86 Proceedings, Vol. 20, No. 4, p. 143-150*, 1986.

[KFYRTK10] S. Kanamori, K. Fujiwara, T. Yoshinobu, B. Raytchev, T. Tamaki, K. Kaneda. Physically-Based Rendering of Rainbows Under Various Atmospheric Conditions. *Computer Graphics and Applications (PG), p. 39-45*, 2010.

[LA06] Clifford Lindsay and Emmanuel Agu. Physically-Based Real-Time Diffraction Using Spherical Harmonics. *Proceedings of the International Symposium on Visual Computing*, 2006.

[Lux] LuxRender, http://www.luxrender.net/en_GB/index Accessed 12/29/11.

[Max] Maxwell Render, http://www.maxwellrender.com/ Accessed 12/29/11.

[MKKLYK07] Shihua Ming, Jung-A Kim, Kyung-kyu Kang, Xianji Li, Sung-yul Yim, and Dongho Kim. Realistic Illumination Model and Caustics Generation Method for Real-time Stained Glass Rendering. *Lecture Notes in Computer Science, 2007, Volume 4563/2007, p. 80-87*, 2007.

[Mus89] F. Kenton Musgrave. Prisms and Rainbows: A Dispersion Model for Computer Graphics. *Proceedings of Graphics Interface '89, p. 227-234*, 1989.

[NND96] T. Nishita, E. Nakamae, and Y. Dobashi. Display of Clounds and Snow Taking Into Account Multiple Anisotropic Scattering and Sky Light. *Proceedings of SIGGRAPH '96, p. 379-386*, 1996.

[RBA09] Michal Radziszewski, Krzysztof Boryczko, and Witold Alda. An Improved Technique for Full Spectral Rendering. *Journal of the International Conferences in Central Europe on Computer Graphics, Visualization, and Computer Vision (WSCG)*, 2009.

[Shi90] Peter Shirley. A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes. *Proceedings on Graphics Interface '90, p. 205-212*, 1990.

[SKP05] Musawir Shah, Jaakko Konttinen, Sumanta Pattanaik. Caustics Mapping: An Image-Space Technique for Real-Time Caustics. *IEEE Transactions on Visualization and Computer Graphics*, 2005.

[SM92] Brian Smits and Gary Meyer. Newton's Colors: Simulating Interference Phenomena in Realistic Image Synthesis. *Proceedings of the Eurographics Workshop on Rendering, p. 185-194*, 1992.

[Spe86] Spencer W. Thomas. Dispersive Refraction in Ray Tracing. *The Visual Computer, Vol. 2, No. 1, p. 3-8*, 1986.

[Sta99]Jos Stam. Diffraction Shaders. *SIGGRAPH '99 Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, p. 101-110*, 1999.

[STI08] Peter Sikachev, Ilya Tisevich, and Alexey Ignatenko. Rendering Smooth Spectrum Caustics on Plane for Refractive Polyhedrons. *18th International Conference on Computer Graphics GraphiCon '08, p. 172-176*, 2008.

[WD06] Chris Wyman and Scott Davis. Interactive Image-Space Techniques for Approximating Caustics. *ACM Symposium on Interactive 3D Graphics and Games, p. 153-160*, 2006.

[WN09] Chris Wyman and Greg Nichols. Adaptive Caustic Maps Using Deferred Shading. *Computer Graphics Forum 28 (2), p. 309-318*, 2009.

[WS03] M. Wand and W. Straßer. Real-Time Caustics. *Proceedings of Eurographics, Vol. 22, No. 3, p. 611-620*, 2003.

[WTP00] Alexander Wilkie, Robert Tobler, and Werner Purgathofer. Raytracing of Dispersion Effects in Transparent Materials. *WSCG 2000 Conference Proceedings, p. 200-207*, 2000.

[Wym08] Chris Wyman. Hierachical Caustic Maps. *ACM Symposium on Interactive 3D Graphics and Games, p. 163-171*, 2008.

[YLY07] Xuan Yu, Feng Li, and Jingyi Yu. Image-Space Caustics and Curvatures. *Computer Graphics and Applications, 2007, p. 181-188*, 2007.

[ĎK05] Roman Ďurikovič and Ryou Kimura. Spectrum-Based Rendering Using Programmable Graphics Hardware. *SCCG '05 Proceedings of the 21st Spring Conference on Computer Graphics, p. 233-236*, 2005.