Ray Tracing Refraction in Hardware

by

Chris Trendall

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Ray Tracing Refraction in Hardware

Chris Trendall

Master of Science

Graduate Department of Computer Science

University of Toronto

2000

Graphics hardware has begun to exhibit more general mathematical functionality, suggesting that a wider range of mathematical operations can be implemented in hardware. In order to explore what can and cannot be implemented in current hardware and to study necessary hardware functionality for even more general calculations, single interaction ray – tracing is studied. The image created by refraction of light from a time – dependent height field is calculated in real – time using graphics hardware. Many of the physical parameters of the scene can be interactively modified. This demonstrates that non – trivial calculations can be performed in hardware at interactive rates.

Acknowledgements

I would like to thank James Stewart for his kind and helpful suggestions, and most of all for allowing me the latitude to explore different areas during the past year or so. In addition, I would like to thank him for his wonderful suggestion to work in France at iMAGIS for a term. This was a very exciting and stimulating experience.

In Toronto, the environment created by all the lab members here at DGP was a very comfortable and friendly one, and I would like to thank all the members of the 'DGP team' for their friendly disposition, general helpfulness, and comraderie.

In France the environment was equally warm, friendly, and stimulating. I would like to thank all the people at the iMAGIS lab for making my stay there so enjoyable. I hope that next time I will not be so pressed for time so that we can interact more.

Last, but certainly not least, I would like to thank Diana for her patience and gentle chiding when I was frantic, her encouragement when I saw nothing but impossibility, the interesting conversations about science that we had, the wonderful meals, hiking, skiing and most importantly, the wonderful joggled lintels that we shared. This was a time that I shall remember always.

Contents

Chapter 1

Introduction

1.1 Motivation

This thesis demonstrates a non–trivial calculation using graphics hardware in order to explore what can be achieved with current hardware, and what should be implemented in hardware in the future. The computational goal is to develop a model to create interactive photo–realistic renderings of the image caused by refraction of light across a surface modelled as a time–dependent height field, using currently available graphics hardware. By studying such a problem, it is hoped that light will be shed on the existing capabilities of hardware, on possible computational models to leverage these abilities, and on directions for the development of a suite of mathematical operations that are useful for implementing algorithms for calculations that are of interest to the graphics community. It is also hoped that the usefulness of such algorithms for *general mathematical calculations* will be shown: that is, that graphics hardware is not just for images any longer.

1.2 Photo–Realism and Interactivity

The computational burden of photo–realistic rendering has created a wide range of approaches toward image generation. At one end of the scale, *hardware–based rendering* can create pseudo–realistic images at interactive rates by using sophisticated graphics pipelines, while at the other end radiosity and ray tracing solutions create highly detailed and accurate images, but use large amounts of computational time.

By hardware–based rendering, we mean the use of algorithms implemented in hardware in order to perform calculations which aid in the display of images. At first glance, this is a fairly mundane definition. However, an examination of the literature shows that, historically, graphics hardware has been used for operations — such as scan conversion — which are directly related to the display of the final image, or devoted to algorithms which support a particular model or representation, such as the polygonal model.

Hardware pipelines were originally focussed on the geometric processing stage: model/view transformations, (surface) lighting, perspective transformation, clipping, and scan conversion. In recent years, more attention has been paid to per–pixel operations such as texturing, depth testing, and blending. As a result, pipelines are beginning to exhibit functions more akin to signal processing, and various complex lighting and shadowing effects have been devised that leverage these newer capabilities. These effects work in real–time or near real–time, and thus the range between slower photo–realistic and faster non–realistic image generation is narrowing.

As the number of lighting effects that can be computed efficiently in hardware increases, it is tempting to think about the set of lighting effects that is desirable to calculate, and the set of operations that is desirable to implement in hardware. However, to do so would be to interpret the term "hardware rendering" in a very limited sense: There are many types of calculations which might occur in order to display an image, and, in order to achieve interactive photo–realism, it is just as necessary to quickly perform these calculations as it is to quickly perform the image–displaying calculations which occur later in a real–time algorithm.

Thus the question of sets of operations becomes much larger: What set of operations would be desirable to have implemented in hardware in order to aid in the calculation and display of images? This is clearly a question which depends upon many things, such as the phenomena that are of interest to model, the models that are available to use, and the techniques for implementing those models. In order to begin to answer this question, one must explore subsets of the domain of enquiry.

The area explored in this work is still strongly image–related, but the calculations are image–based rather than geometrically based. The approach used is to leverage the signal processing capabilities of the current graphics hardware to perform some of the more complex mathematical operations necessary to achieve detailed lighting effects. In particular, refraction of light through a time–dependent height field is calculated. This is only one of possibly many examples to demonstrate the kind of mathematical functionality that current graphics hardware possesses. In addition, and perhaps more importantly, this work explores the possibility of treating lighting fundamentally as a *signal processing* problem, rather than as a geometric problem, or a sampling problem.

Chapter 2

Previous Work

2.1 Hardware Rendering

There are many examples of lighting effects that have been computed with algorithms in hardware, but few examples of other types of calculations. What follows is a summary of hardware-based calculations, with particular attention to the techniques that are relevant to this thesis.

Phong [36] or Blinn–Phong reflection models [6] are commonplace in current hardware implementations. More recently, Heidrich [24, 26, 21] has detailed methods to implement a wider variety of reflection models using multi–pass methods on existing hardware. His approach is more in line with the work in this thesis: A set of carefully chosen mathematical operations can be combined in order to make a much broader range of calculations.

Heidrich shows how the Torrance–Sparrow [43, 44] isotropic and the Banks [3] anisotropic reflection models can be implemented using operations currently available in hardware. In addition, he suggests some basic operations which could be implemented in hardware to support a flexible per–vertex lighting model [32, 21]. Heidrich also shows how specifying a 'normal map' or a texture map containing the normals to the surfaces of objects leads to the ability to efficiently calculate Blinn–Phong lighting using only the imaging part of the graphics pipeline.

In the same spirit, Heidrich also suggests a decoupling of illumination from surface geometry [23, 21]. In this scheme, geometry information is coded into texture maps which are used to look up illumination information from light fields or environment maps. This doesn't really decouple geometry from illumination, but suggests a different representation for geometry and a different approach to illumination — the geometry becomes a transfer function in a signal processing approach.

Haeberli and Segal [16] show how texture mapping can be used to implement anti–aliasing, volume rendering, air–brushes, movie display, contouring, colour interpolation in non–RGB colour spaces, Phong shading, environment mapping, and 3D half-toning. Used in these manners, texture mapping can be thought of more generally as a transfer function between geometry and image space. This is a powerful concept which leads naturally to treating image generation as a signal processing problem, and developing more signal processing functionality in hardware. Both the signal processing approach and the development of the mathematical functionality of the graphics card exhibited by Haeberli and Segal and Heidrich are along the lines of the thesis of this work.

McCool and Heidrich [32] proposed a set of hardware extensions to better support programmable shading through texture mapping. The authors identify common interactive rendering tasks for phenomena that are considered important by the graphics community, and abstract a set of operations which provide the computational power to render these phenomena in real time. The operations selected are done so to provide the power to perform the computations deemed important, while providing the flexibility to be able to combine them in new ways to perform other types of computation. This is one of the major steps forward in developing graphics card functionality. However, it is concerned with lighting only rather than with more general calculations.

Voorhies *et al.* [45] show how a cubic environment map can be used with unnormalized reflection vectors to generate surface reflections if the hardware admits a per–pixel division. At the time, they were implementing this in hardware for a high–end workstation.

Reflections from planar mirrors have been rendered using a multi–pass algorithm [13, 12, 14]. In this algorithm, the reflecting surface is rendered into the stencil buffer, the scene is rendered omitting the light from the reflecting surface, the viewpoint is shifted to behind the reflecting surface, and the scene is rendered again but only over the previously masked pixels. This leverages the knowledge that the angle of incidence is equal to the angle of reflection, and that these angles are constant over a planar surface.

Refractions from light transmitted through planar surfaces from a single incident angle have also been treated by Diefenbach *et al* [13, 12]. The treatment is similar to that of planar reflection, except that the refracting surface is considered to be infinitely thin, which results in a rotation of the viewpoint by an amount given by Snell's law. This results in distortion due to the different orientation of the image plane on the second pass, which the author corrects for. However, real objects have finite thickness, and light which is refracted by a transparent object with parallel incident and exit surfaces should shift the position of the light from the object, not rotate it.

Reflections from curved surfaces have been generated using environment maps [6]. Spherical environment maps, which have been the standard until recently, are suitable for reflection from a curved surface for a single viewing direction. The much more flexible parabolic parameterization [25, 21] allows multiple viewing directions.

Shadows have also been calculated using hardware. Shadow volumes [11, 12] and shadow maps [47, 37] have been implemented in hardware [1, 34], and new algorithms for hardware implementation have been proposed [21].

Another feature which has been implemented in hardware is bump mapping [5]. Although dedicated algorithms for hardware bump mapping have been designed, [15, 33, 35] algorithms which use different hardware features on each pass of a multi–pass algorithm have also been designed [7, 21]. This second approach is somewhat more interesting in relation to this work as it utilizes more fundamental operations to build more complex operations. Blythe *et al.* [7] have also developed many other effects using graphics hardware.

Light sources have been simulated using currently available hardware for interactive rendering by Heidrich [22, 21], in one of his many contributions to the use of hardware graphics operations for computation. In those works, a discretely sampled version of a light source is stored in the same manner as in image–based rendering, and light in the scene is simulated by using projective texturing in an extension to Segal's [37] approach.

In an application which is suggestive of the possibility of using graphics hardware to do more general mathematical calculations, Heidrich [27] shows how the line integral convolutions of a vector field introduced by Cabral *et al.* [8] and further developed by Stalling and Hege [40] can be fully implemented in current hardware. Although a line integral is a general mathematical operation, these research was originally focussed on the *visualization* of the vector field, rather than the merits of performing the calculation itself in hardware. It is exactly this avenue that would be interesting to explore — more generalized mathematical calculations.

In one of the few uses of graphics hardware for mathematical calculations that are less directly connected with image synthesis, Hoff *et al.* [28] leverage z buffering capabilities to calculate Voronoi diagrams. Their approach is to compute a mesh for each Voronoi site whose height represents the distance from that site. Rendering each mesh in a different colour, using depth buffering, leaves coloured Voronoi sets in the colour buffer and distances in the depth buffer.

In another use of graphics hardware for a non–graphics algorithm, Lengyel *et al.* [31] perform real–time robot motion planning using rasterizing hardware. However, in that work the graphics hardware is used for very little — merely to exclude positions to which the robot cannot move by rendering obstacles. The rest is done in software.

2.2 Refractive Caustics

All the work discussed in this section is neither interactive, nor achieved using hardware–based rendering. However, this is an overview of the treatment of refraction in computer graphics and as before, only works directly relevant to this thesis will be elaborated upon.

Stam [41] used a probabilistic approach and the wave theory of light to calculate caustics from a randomly varying surface which is correlated in time. The textures are calculated offline, and then rendered in sequence to animate them. This is appropriate for situations in which the viewer is unable to correlate the refracting surface with the refracted caustic since there would be no direct deterministic connection. In this thesis, we attempt to make the deterministic calculation, and to do so at interactive rates.

Arvo [2] used backward ray tracing to calculate the position of photons emitted from a light source incident on an object in a scene. In order to calculate the irradiance, he used illumination maps – texture maps that count the number of photons incident in a fixed area. This is clearly not an interactive procedure, and many photons must be used to eliminate noise. Interestingly, he is using a texture map to accumulate illumination. This is similar to the work in this thesis, in that the discretization of the frame buffer is mapped to the discretization of the problem.

Many variants of Arvo's approach have been proposed:

- Heckbert [18] improved the technique by adaptively subdividing the area, making each traced photon more efficient in its contribution to the total irradiance.
- Chen *et al.* [9] and Collins [10] used a fixed illumination map, but spread energy from photon hits to nearby surface elements to smooth noise. A 'spreading' function is also used in this thesis.
- Jensen [30] used a photon map: a kd–tree which stores the position, energy, and incoming direction of each photon. To compute the irradiance at a point, the algorithm searches the kd–tree for the N closest photons, normalizing by the area

of the circle of radius equal to that of the smallest sphere which encircles the photons.

Heckbert and Hanrahan [19] leveraged the spatial coherence of polygonal objects by reflecting and refracting beams of light from the visible surfaces in the view frustum. A tree structure is used to store the subdivided beams and can grow quite large in highly tessellated environments. However, for simple environments this method will outperform ray tracing.

Watt [46] calculated the caustics on the bottom of a swimming pool using backward beam tracing. In this scheme, 'caustic polygons' are created by casting rays from the vertices of each surface or 'specular' polygon to the light source, calculating the direction of refraction using the surface normals, and then testing the beam defined by the refracted rays for intersection with objects in the scene. The scene or 'diffuse' polygons affected are recorded, and the process continues for another specular polygon. At the end of this process, the rendering pass begins. A visible diffuse polygon is checked to see if a caustic polygon is associated with it, and, if so, the intensity of the caustic polygon is added to the diffuse component of the reflection of the diffuse polygon. Unfortunately, the formulas used by Watt for the 'intensity' and 'energy' do not correspond to radiometric quantities; energy as he has defined it is not actually conserved. Although Watt gives no frame rates for the images that he depicts, it seems quite unlikely that this method runs in real time.

Both Watt's approach [46] and Heckbert and Hanrahan's approach would involve rendering multiple polygons and blending them in order to create the refractive caustic, which would require on the order of n passes if the surface was comprised of n polygons. This is a result of their discretization. In this work, we avoid this problem by making a continuous approximation to the spreading of light after refraction, which leads to an integral that can be discretized.

Chapter 3

Mathematical Capabilities of Graphics Hardware

Modern raster graphics implementations typically have a number of buffers such as an accumulation buffer, a depth buffer, a stencil buffer, and one or more general purpose rendering buffers. Current buffers usually have a depth of 32 bits per pixel or more, and there can be various schemes for allocating different numbers of those bits to elements or components within the pixel. Common element combinations are: red, green, and blue (RGB), used to represent colour; red, green, blue, and alpha (RGBA), used to represent colour; and opacity or luminance, which is a single element configuration used to represent grayscale, for example [48]

This flexibility allows for an interpretation of an image as a discrete vector field or as a height function defined on a discrete rectangular domain in the xy plane. The luminance value of a pixel might represent the value of the function while the position of the pixel in the image represents the position in the xy plane. Alternatively, an RGB or RGBA image could represent a three or four dimensional vector field defined over a subset of the plane.

The beauty of this kind of interpretation is that operations on an image are highly parallelized and calculations on entire functions or vector fields can be performed very quickly in graphics hardware. This approach treats the graphics hardware like a SIMD (Single Instruction, Multiple Data) machine [32], and as will be shown there is some range in the types of operations that can be performed.

One of the drawbacks is that only limited precision can be expected due to the limited depth of the buffers; this can be a problem particularly for vector fields in which components often can be assigned only eight or ten bits. The other major drawback is the lack of signed arithmetic. As a result, much finagling of scaling and biasing coefficients must be done in order to simulate signed arithmetic. This is particularly troublesome as the pipeline clamps the element values to $[0, 1]$ in a number of places.

For concreteness, the functionality of OpenGL will be considered, along with some of the experimental extensions offered on some SGI machines. The functions mentioned are all in the rasterization and per–fragment areas of the pipeline; in particular the *OpenGL 1.2 imaging subset* is used heavily [48]. This is in contrast to the geometry section of the pipeline which had been the core of hardware rendering before texture mapping became so prevalent: All operations discussed in this thesis are performed by moving rectangles of pixels through the imaging pipeline.

3.1 Imaging Pipeline

The imaging pipeline of OpenGL 1.2 is shown in Figure 3.1 [48]. The most important operations to this thesis are the pixel transfer operations of which the imaging subset is a component. The pixel transfer section of the imaging pipeline is shown in detail in Figure 3.2 [48].

Some of these operations are always enabled but most can be enabled and disabled; they operate on pixel rectangles which are drawn to a buffer from host memory, copied between buffers, read from a buffer to host memory, copied into a texture map from

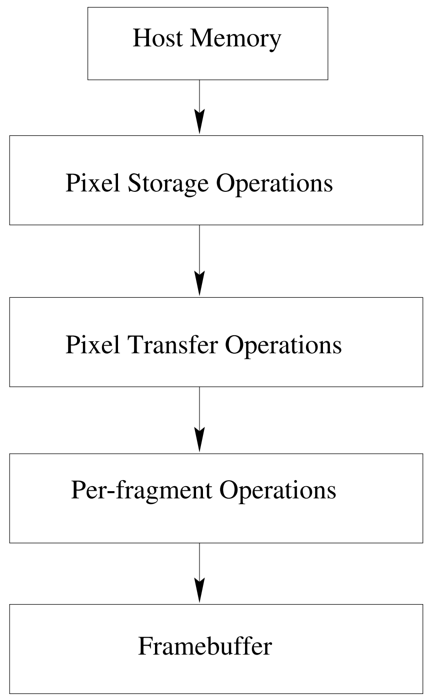

Figure 3.1: OpenGL 1.2 Imaging Pipeline

either host memory or the framebuffer, or read from a texture map into host memory.

Only the functionality which is used in this work is discussed here; there are many other hardware features the power of which can also be leveraged. In addition to the imaging subset of OpenGL 1.2, the pixel texture extension [29] is both used and described here. The imaging subset is an optional set of operations; vendors are not required to implement them. However, if the subset is implemented, *all* the functionality must be implemented [38].

The following is a description of the hardware features used in this thesis [48, 38].

3.1.1 Colour Tables

Colour Lookup Tables allow for one dimensional mapping from a pixel component value to another pixel component value. Each pixel component (R,G,B, and A) can be independently mapped. The domain and range of the mapping are limited to one dimension,

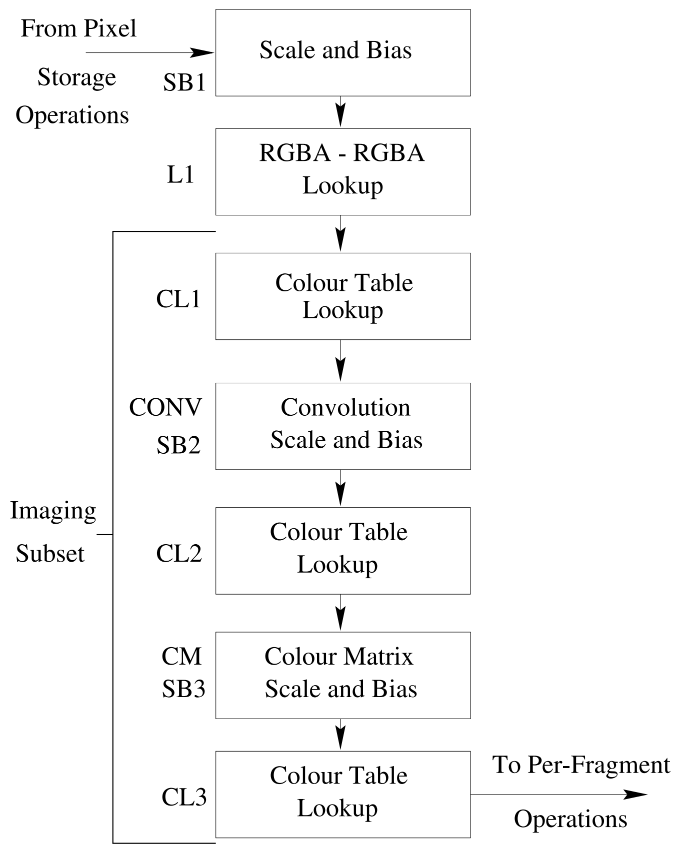

Figure 3.2: Pixel Transfer Operations. Values are clamped to $[0, 1]$ upon leaving this part of the pipeline

and the mapping is calculated in hardware by multiplying the value of the pixel component by an integer one less than the table size, truncating, and using this integer to index the new value. (This corresponds to what would be the GL_NEAREST mode in texture mapping.)

There are several places in the imaging pipeline where colour tables can be used, and different tables can be used on each of the elements in a pixel group at each such place. This allows the application of a wide range of many–to–one functions, although in one dimension as mentioned. Colour Tables cause the resulting pixel element values to be clamped to the range $[0, 1]$.

3.1.2 Convolution

A discrete convolution is a linear combination of neighbouring function values. The weights of each of the neighbours in the sum is determined by a *kernel*. A two–dimensional convolution computes

$$(f * K)_{i,j} = \sum_{m=-W}^{W} \sum_{n=-H}^{H} K_{m+W,n+H} f_{i+m,j+n},$$

where K is the kernel of size $2W + 1$ by $2H + 1$, f is the function being convolved, and $f * K$ is the resulting function.

The imaging subset supports convolutions in one and two dimensions, and each element in a pixel group can be convolved with a different filter during a single pass. Convolutions are an important feature in signal processing as they define a set of linear filters. Filter sizes are limited depending on the implementation. For example, on the IMPACT and InfiniteReality2 renderers the maximum filter size is 7×7, while on other SGI renderers, the maximum filter size is 11×11.

Larger filters can be obtained by dividing the larger filter into many pieces of hardware–supported size, making multiple passes, and accumulating the results. This can, of course, take much longer, and the accumulation buffer must be used because of the extended precision needed with the multiple passes.

3.1.3 Colour Matrix

The Colour Matrix allows each of the RGBA outputs of this stage of the imaging subset to be a linear combination of the RGBA inputs. The colour matrix cannot be disabled, but it can be set to the identity matrix. In OpenGL, colour matrices can be manipulated in the same manner as other OpenGL matrices.

3.1.4 Scaling and Biasing

At several stages of the imaging pipeline, arbitrary scale and bias factors may be specified. These can be used to prevent loss of data due to $[0, 1]$ clamping, for example.

In addition to these features, a histogram function and a minmax function are specified in OpenGL 1.2 to be part of the imaging subset. They are not used in this work. One could imagine, however, that the minmax feature might be used to maximize the "dynamic range" of the calculation: The minmax feature could be used to determine the range of results from a certain calculation, and the calculation could be repeated with new scale and bias parameters such that the range of new results occupies all of $[0, 1]$, rather than just a limited subset.

3.1.5 Pixel Textures

Pixel texturing provides a powerful tool for mapping between a domain of up to four dimensions and a range of up to four dimensions [29]. This technique is commonly called a lookup table or LUT, and provides the ability to implement a large range of functions and compositions of functions. The purpose of creating this extension was to provide a method to texture an image on a per–pixel basis rather than just the per–vertex basis which is normally used. This per–pixel computation works by interpreting the RGBA component values of the pixels in the pixel rectangle as co–ordinates in a texture map. The values must be in the range $[0, 1]$ and the texture map must have the same number of dimensions as elements in the pixel group: a four dimensional texture for RGBA, three dimensional for RGB, two dimensional for LUMINANCE_ALPHA, or one dimensional for LUMINANCE. Each element in the pixel group is multiplied by the size of the associated dimension of the texture minus one, and the indexed texel (or some average of nearby texels, depending on texturing mode) is retrieved. The indexed texel then replaces the indexing pixel RGBA values in the pixel rectangle.

This extension is not part of OpenGL 1.2, but exists on HIGH IMPACT, MAXIMUM IMPACT, and the higher end Octane renderers. Unfortunately, it is not part of the InfiniteReality series of renderers [29].

3.1.6 Blending

Blending is part of the per–fragment operations that include the Pixel Ownership Test, the Scissor Test, the Alpha Test, the Stencil Test, the Depth Buffer Test, Dithering, and Logical Operations. It allows the combination of the incoming fragment's RGBA values with the RGBA values stored in the framebuffer at the destination of the pixel rectangle. The default equations are

$$\begin{aligned}
R_{Final} &= R_{Fragment} * S_{Blend,R} + R_{Buffer} * D_{Blend,R}, \\
G_{Final} &= G_{Fragment} * S_{Blend,G} + G_{Buffer} * D_{Blend,G}, \\
B_{Final} &= B_{Fragment} * S_{Blend,B} + B_{Buffer} * D_{Blend,B}, \\
A_{Final} &= A_{Fragment} * S_{Blend,A} + A_{Buffer} * D_{Blend,A},
\end{aligned}$$

where $C_{Final} = (R, G, B, A)_{Final}$ is the value of the component left in the framebuffer after the operation, $C_{Fragment} = (R, G, B, A)_{Fragment}$ is the value of the component of the incoming fragment, $C_{Buffer} = (R, G, B, A)_{Buffer}$ is the value of the component already in the buffer, and $S_{Blend,C}$ and $D_{Blend,C}$ are source and destination blending factors. These blending factors can take on a variety of values, from the colour of the source or destination, to the alpha of the source or destination, to a user defined constant value. In addition, several other blending operations are possible other than the addition shown above such as subtraction, min, and max, although they are not used in this thesis.

3.2 Mathematical Operations

The operations which are detailed below are all used in the refraction demonstration: discrete derivatives of scalar functions, dot products of scalar functions, normals to a height field, signed products of scalar functions, signed sums of scalar and vector functions, compositions of scalar and vector functions, and convolutions. In all cases, scalar and vector functions are defined on a discrete domain.

3.2.1 Convolutions

One and two dimensional convolutions are directly implemented in hardware. The size of the kernel is typically limited on current implementations to 5×5, 7×7, or 11×11. Separable kernels (i.e. those in which $K_{i,j} = a_i \times b_j$ for some vectors a and b) are supported and encouraged, since convolutions with separable kernels are faster. For larger kernels, the kernel must be split into smaller, hardware–supported sizes and multiple passes must be made. This approach was taken by Soler [39] in generating soft shadows by hardware convolution.

3.2.2 Derivatives

Discrete derivatives of height fields can be taken either by using convolution [48] or by blending in the accumulation buffer [7].

3.2.3 Normals to a Height Field

Let us define the domain of the continuous height field as

$$\Omega \;=\; [0, M\Delta x] \times [0, N\Delta y],$$

where $N, M \in \mathbb{N}_0$ and $\Delta x, \Delta y \in \mathbb{R}$. Then the height field is defined by

$$h : \Omega \to \mathbb{R}.$$

A vector orthogonal to $h(x, y)$ is given by

$$\vec{N} = \left(-\frac{\partial h(x,y)}{\partial x}\ , -\frac{\partial h(x,y)}{\partial y}\ , 1 \right)$$

and the unit normal, $\widehat{\mathbf{N}}$, is given by

$$\widehat{\mathbf{N}} \;=\; \frac{\vec{N}}{\mid \vec{N} \mid} = \frac{(-\frac{\partial h(x,y)}{\partial x}\ , -\frac{\partial h(x,y)}{\partial y}\ , 1)}{\sqrt{(\frac{\partial h(x,y)}{\partial x}\ )^2 + (\frac{\partial h(x,y)}{\partial y}\ )^2 + 1}}.$$

For the discretized version, let patch Ω_{ij} (for $0 \leq i < M$ and $0 \leq j < N$) of the domain be

$$\Omega_{i,j} = [i\Delta x, (i+1)\Delta x] \times [j\Delta y, (j+1)\Delta y],$$

the corresponding patch on the height field be

$$A_{ij}(x, y) = h(\Omega_{i,j}),$$

and a corresponding point at the corner of a patch be

$$h_{ij} = h(i\Delta x, j\Delta y).$$

We assume that $A_{ij}(x, y)$ can be approximated by a portion of a plane with

$$\frac{\partial A_{ij}}{\partial x} \approx \frac{(h_{i+1,j} - h_{i,j})}{\Delta x} \qquad \text{and} \qquad \frac{\partial A_{ij}}{\partial y} \approx \frac{(h_{i,j+1} - h_{i,j})}{\Delta y}.$$

Define $\Delta_i h_{ij} = h_{i+1,j} - h_{i,j}$ and $\Delta_j h_{ij} = h_{i,j+1} - h_{i,j}$. Then the discretized version for the unit normal to $A_{ij}(x, y)$ is given by

$$\widehat{N}_{ij} = \frac{\vec{N}_{ij}}{\mid \vec{N}_{ij} \mid} = \frac{(-\Delta_i h_{ij}\ , -\Delta_j h_{ij}\ , 1)}{\sqrt{(\Delta_i h_{ij}\ )^2 + (\Delta_j h_{ij}\ )^2 + (\Delta x)^2}},$$

where it is assumed for brevity that $\Delta x = \Delta y$.

Thus $\widehat{N}_{ij}$ is a function of the discrete partial derivatives $\Delta_i h_{ij}$ and $\Delta_j h_{ij}$ and the constant Δx. If $\Delta_i h_{ij}$ and $\Delta_j h_{ij}$ are in two of the color components of an RGBA image, as might happen after calculating the derivatives of the height field using convolution, pixel texturing can be used later in the pipeline to look up a pre–computed normal. A three dimensional normal has only two degrees of freedom (due to the restriction that the sum of the squares of the components must equal one) so a two–dimensional texture map may be used for the lookup [20].

Since $\Delta_i h_{ij}$ and $\Delta_j h_{ij}$ can be signed, the results from the discrete partial derivative calculation must be scaled and biased so that $\Delta_i h_{ij}\ , \Delta_j h_{ij}\ \in [0, 1]$. The scaling and biasing parameters should either be known *a priori* as a result of estimating the upper and lower bounds on normals, or multiple lookup tables can be calculated and stored in host memory awaiting a dynamic determination of the ranges of the partial derivatives in order to select the appropriate table. In any case the contents of the lookup table or texture are dependent on these scaling and biasing parameters. In addition, the *contents* of the texture must be scaled and biased into the interval $[0, 1]$ and any subsequent calculations with these scaled and biased normals must carefully take the scaling and biasing into account. The details of the scaling and biasing will be discussed in Chapter 5.

3.2.4 Dot Products

The dot products of each vector in a vector field with up to four different vectors can be calculated in the same pass by loading the vectors into the Color Matrix. Application of the matrix then multiplies the matrix vectors component–wise with those in the field, and sums the products. The elements of the colour matrix are not clamped — signed values can be used, for example, but the values in the pixel rectangle must be unscaled and unbiased before the dot product. The result must be scaled and biased again in order to preserve its sign and value as it will be clamped to $[0, 1]$ later in the pipeline. The details of this procedure will be discussed in Chapter 5.

3.2.5 Arithmetic on Scaled and Biased Values

Since signed arithmetic isn't supported at all points of the imaging pipeline, extra precautions must be taken to provide this functionality. In addition, values are clamped to $[0, 1]$ in many places, which forces the programmer to carefully scale and bias values to keep them in this range.

A constant scalar or vector can be added to or subtracted from a scalar or vector function by using the bias facility. As shown in Figure 3.2, there are three places at which this can occur. OpenGL allows one to scale and bias by an arbitrary positive or negative number.

Let $\widetilde{f}$ represent a function f that has been scaled by α and biased by β. That is,

$$\widetilde{f} = \alpha f + \beta.$$

A function that has been scaled and bias is said to have been *adjusted*. We will assume that the same scale and bias are applied to all functions, since the scale and bias will be chosen from our expectation of the range of the results of arithmetic combinations of these functions.

In what follows, it is shown how to perform basic arithmetic operations on functions and constants such that the result is correctly adjusted. The distinction between functions and constants is made because constants are often faster to use in the hardware pipeline: Consider, for example, that adding a constant can be done (cheaply) by using a single bias parameter or (expensively) by blending a pixel rectangle of constant value.

Sums and Differences

To sum an adjusted function, $\widetilde{f}$, with an unadjusted constant, c, the programmer must pre-multiply c by the same scale factor which was applied to the function:

$$\begin{aligned} \widetilde{f+c} &= \alpha\ (f+c)+\beta \\ &= \alpha\ f+\beta+\alpha\ c \\ &= \widetilde{f}+\alpha\ c. \end{aligned}$$

To perform this in hardware, it is sufficient to set the bias to αc at some point in the pipeline and to send $\widetilde{f}$ through the pipeline. Since one can independently choose the scale and bias values for each of the colour components and alpha, vector sums can be treated in the same manner.

In summing two functions, $\widetilde{f}$ which is adjusted and g which is not, we have the same equation:

$$\widetilde{f+g}=\widetilde{f}+\alpha\ g.$$

To perform this operation in hardware, we assume that $\widetilde{f}$ in already in the framebuffer and then send g through the pipeline. We must scale g by α and use blending with an additive blend function. The scaling can be done at one of the scale and bias points, or g can be scaled by a constant user–specified value in the blend equation. Subtraction works the same way with a subtractive blending function.

If both quantities, $\widetilde{f}$ and $\widetilde{g}$, are already adjusted, we sum them as follows:

$$\begin{aligned} \widetilde{f+g} &= \alpha(f+g)+\beta \\ &= \alpha f+\beta+\alpha g+\beta-\beta \\ &= \widetilde{f}+\widetilde{g}-\beta. \end{aligned}$$

Thus β must be subtracted from the sum either before or after additive blending, or must be added to the difference after subtractive blending. This can be done at a scale and bias point.

The sum and difference of vector functions works in the same manner, except all components must be scaled or biased equally.

Products

Multiplication of an adjusted function $\widetilde{f}$ by a constant c is done as follows:

$$\begin{aligned}\widetilde{cf} &= \alpha\,(c\,f)+\beta \\ &= \alpha\,c\,f+\beta\,c+\beta-\beta\,c \\ &= c\,\widetilde{f}+(1-c)\,\beta.\end{aligned}$$

This can be done in the hardware by first copying to the framebuffer a pixel rectangle in which each component is set to the bias value β. The source blend factor is then set to c and the destination blend factor is set to $1-c$, and $\widetilde{f}$ is sent through the pipeline. Alternatively (and this depends upon where in the pipeline $\widetilde{f}$ becomes available) we can scale $\widetilde{f}$ by c and bias it by $(1-c)\beta$.

This technique also works for vectors. Each colour component of the constant can be independently modified so as to scale the components of the vector field by different amounts, if wanted.

Multiplication of an adjusted function $\widetilde{f}$ by an unadjusted function g can be achieved in a similar manner. Again we have

$$\widetilde{fg} = g\,\widetilde{f}+(1-g)\,\beta.$$

Assuming that $\widetilde{f}$ is in the RGB components of the pixel rectangle and that g is in the ALPHA component (which can always be achieved) then a blend as above with a source blend factor of ALPHA and a destination blend factor of (1 – ALPHA) achieves the proper result.

This analysis assumes that the adjusted functions are sent together down the pipeline, which requires one to store β beforehand in the framebuffer. If, on the other hand, the framebuffer already stores $\widetilde{f}$, the β pixel rectangle is sent down the pipeline (in RGB) with g in the ALPHA component.

The final case is the multiplication of a two adjusted scalar functions, $\widetilde{f}$ and $\widetilde{g}$. This is performed using the following equation:

$$\begin{aligned}
\widetilde{fg} &= \alpha\, fg + \beta \\
&= \alpha\, fg + \beta(f+g) + \frac{\beta^2}{\alpha} - \beta\,(f+g) - \frac{\beta^2}{\alpha} + \beta \\
&= \frac{1}{\alpha}\,(\alpha^2\, fg + \alpha\beta\,(f+g) + \beta^2) - \frac{\beta}{\alpha}\,(\alpha \cdot (f+g) + \beta - \alpha) \\
&= \frac{1}{\alpha}\,(\widetilde{f} \cdot \widetilde{g}) - \frac{\beta}{\alpha}\,((\widetilde{f+g}) - \alpha).
\end{aligned}$$

If we assume that $\tilde{g}$ is already in the framebuffer, then the procedure is as follows:

BEGIN:

Pass 1: Calculate $\widetilde{f+g}$ (see above)

Pass 2: Additively blend $\tilde{f}$ with $\tilde{g}$ to give $\tilde{f} \cdot \tilde{g}$.

$\tilde{g}$ is the source blend factor, 0 is the destination blend factor.

Pass 3: Bias $\widetilde{f+g}$ by $-\alpha$ and subtractively blend with $\tilde{f} \cdot \tilde{g}$ to give

$\frac{1}{\alpha}\,(\widetilde{f} \cdot \widetilde{g}) - \frac{\beta}{\alpha}\,((\widetilde{f+g}) - \alpha)$.

The blend factor for $\tilde{f} \cdot \tilde{g}$ is $\frac{1}{\alpha}$;

The blend factor for $\widetilde{f+g} - \alpha$ is $\frac{\beta}{\alpha}$.

END

To implement this in hardware, we first calculate the sum of the two adjusted functions $\widetilde{f+g}$ in Pass 1 as explained earlier. (Note that α and β must have been chosen such that $\widetilde{f}$, $\widetilde{g}$, and $\widetilde{f+g}$ are in the range $[0,1]$.)

We then blend $\widetilde{f}$ and $\widetilde{g}$ in Pass 2 so as to give the product $\widetilde{f} \cdot \widetilde{g}$. Note that the result is guaranteed to be in $[0,1]$ since both $\widetilde{f}$ and $\widetilde{g}$ are in $[0,1]$.

Finally, $\widetilde{f+g}$ is biased by $-\alpha$ and this result is blended, using subtractive blending, with $\widetilde{f} \cdot \widetilde{g}$ in Pass 3. Multiplication of two adjusted functions in this manner is quite expensive, requiring three passes through the pipeline.

3.2.6 Compositions of One–Dimensional Functions

Indexing into a table is essentially a many–to–one function, since the value of each pixel component in a pixel rectangle can be mapped independently to an arbitrary value. In the OpenGL imaging pipeline there are three colour tables and one RGBA to RGBA lookup, all of which operate in a similar manner. However, after a colour table has been applied, the pixels will be clamped to $[0, 1]$. Before this happens, a post–colour table scale and bias is allowed. These lookup tables allow for the composition of up to four many–to–one functions in a single pass, with the ability to insert other operations in between, as shown in Figure 3.2.

The colour tables only allow functions that have a one dimensional domain and a one dimensional range. Higher dimensional mappings are implemented using the pixel texture extension. This extension allows us to implement functions that map n dimensions to m dimensions where $n, m \leq 4$.

3.2.7 Summary

The preceding operations are summarized in Table 3.1. Note that since clamping always occurs after CL3 and after Blend, all values must be properly scaled and biased by the end of these operation. Furthermore, clamping also occurs in order to perform colour look-ups, but only if the appropriate CL is enabled. Thus any scale and bias operation indicated in the table can occur at another scale and bias point when there is no clamping between these two points. The scale and bias points in the table were chosen in order to minimize the length of imaging pipeline needed for each operation.

Mandatory hardware operations are shown with black squares, while optional operations are indicated with empty squares. These operations are only optional if the values passing through the the pipeline are in $[0, 1]$ already, and need not be scaled and biased. The composition operation can be comprised of any number of colour table look-ups.

Mathematical Operation	Hardware Operation									
	SB1	L1	CL1	CONV	SB2	CL2	CM	SB3	CL3	BLEND
CONVOLUTION	□			■	□					
DERIVATIVE (CONV)	□			■	□					
HF ORTHOGONAL†	□			■	□					
DOT PRODUCT					□		■	□		
ADDITION/SUBTRACTION										
$\tilde{f} \pm c$	■									
$\tilde{f} \pm g$ ‡								□		■
$\tilde{f} \pm \tilde{g}$ ‡								□		■
MULTIPLICATION										
$\tilde{f}c$	■									
$\tilde{f}g$ ‡										■
$\tilde{f}\tilde{g}$, PASS 1 ‡								□		■
$\tilde{f}\tilde{g}$, PASS 2										■
$\tilde{f}\tilde{g}$, PASS 3								□		■
COMPOSITION		□	□			□			□	

Table 3.1: Summary of mathematical operations and the hardware procedures that can be used to implement them. ■ Mandatory operation. □ Optional operation; used to counteract clamping. † Orthogonal vector to height field; can be normalized if pixel texturing is implemented. ‡ Requires previous pass with no hardware operations enabled to copy constant to framebuffer.

Chapter 4

Mathematics of Refraction

This thesis uses geometric optics to describe light incident on a plane beneath a boundary between two media with differing optical properties. This phenomenon is called refraction: the bending of light at the interface between the two materials. The medium in which the light originates is usually called the incident medium while the second medium is called the transmission medium. In this work, the incident medium is considered to be air, which is modelled as vacuum, and the transmission medium is considered to be water. The model for the water includes terms which describe the particle content; this gives rise to attenuation and diffusion of the light in the water. The curvature of the water surface and the incident direction of the light are assumed to be such that the water doesn't cause any self–shadowing phenomena and the scattering of light is assumed to occur mainly at the air–water boundary.

4.1 Bending of Light at an Interface

The optical property which must differ in order for light to bend is simply that the speed of light must be different in each of the two media, and the parameter which describes the speed of light in a non–conducting medium relative to the speed of light in a vacuum is called the *absolute index of refraction*:

$$n(\lambda) = \frac{c}{v},$$

where n is the absolute index of refraction in a non–conducting medium, λ is the wavelength of light, c is the speed of light in vacuo, and v is the speed of light in the medium.

If the index of refraction of the medium in which the light originates has index of refraction n_i and the index of refraction of the medium into which the light is transmitted has index of refraction n_t then the quantity

$$n_r(\lambda) = \frac{n_i(\lambda)}{n_t(\lambda)}$$

is called the *relative index of refraction.* The amount of bending that light undergoes at the interface is dependent upon the relative index of refraction.

Note that using this model, light travels in straight lines in each medium without attenuation, and that the index of refraction is wavelength dependent. This latter property gives rise to phenomena such as *dispersion* which is most notably illustrated by the wavelength– or colour–dependent bending of white (multi–wavelength) light as it encounters the boundary of a prism.

For conducting media, the situation is slightly different as the conductivity of the medium causes attenuation by absorption. Here the index of refraction is complex:

$$n_C(\lambda) = n_R(\lambda) + i\ n_I(\lambda),$$

were n_C is the complex index of refraction, $n_R \in \mathbb{R}$ describes the speed of light in the medium, and $n_I \in \mathbb{R}$ describes the attenuation or absorption of light in the medium and is proportional to the conductivity of the medium [17].

The bending of light at the boundary between media is shown in Figure 4.1. Here $\widehat{\mathbf{N}}$ is the unit normal to the interface, θ_i is the angle of incidence, θ_t is the angle of transmission, $\widehat{\mathbf{L}}_\mathbf{i}$ is a unit vector pointing in the direction of the incoming light (from the

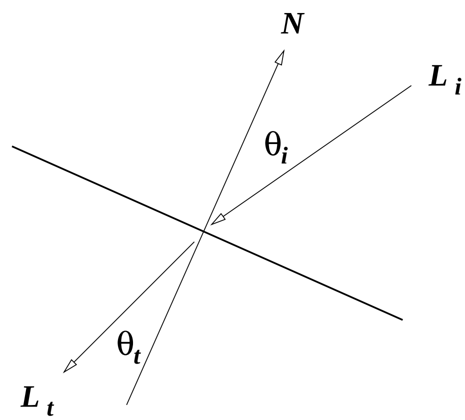

Figure 4.1: Refraction: the bending of light at the interface between two media with differing indices of refraction

source), and $\widehat{\mathbf{L}}_{\mathbf{t}}$ is a unit vector pointing in the direction of the transmitted light after interaction with the interface. The relationship between θ_i and θ_t is given by *Snell's Law*:

$$n_i \sin\theta_i \; = n_t \sin\theta_t \;, \tag{4.1}$$

where the explicit wavelength dependence of the indices of refraction has been dropped for clarity. Thus the direction of the transmitted light is a function of the indices of refraction of the two media and the angle of incidence of the incoming light.

In vector notation, Snell's Law becomes [42]:

$$\widehat{\mathbf{L}}_{\mathbf{t}} \; = \frac{n_i}{n_t}\, \widehat{\mathbf{L}}_{\mathbf{i}} \; + \Gamma\, \widehat{\mathbf{N}} \;, \tag{4.2}$$

where

$$\Gamma = \frac{n_i}{n_t}\, \cos\theta_i \; - \cos\theta_t \;. \tag{4.3}$$

4.2 Radiometry at an Interface

Although geometric optics is used to model light in this work, all of geometric optics may be derived from wave optics. In wave optics, light is a form of a much more general disturbance — *electromagnetic radiation.* The study of the measurement of electromagnetic radiation is called *radiometry*, and since image synthesis concerns itself with exhibiting theoretical values of measured light, some of the quantities of radiometry are of interest.

Radiant flux is the power, or amount of energy per unit time passing through some (possibly fictitious) surface.

Irradiance is the density of radiant flux passing through some surface, and is measured in radiant flux per unit surface area. Irradiance usually refers to incident flux density, while *exitance* or *radiosity*, refers to flux density leaving a surface.

Radiance, denoted by L, is the flux density per unit solid angle passing through some surface which is perpendicular to the direction of the flux.

Intensity is the flux per unit solid angle. It is, for example, the radiance on the surface of a sphere of radius $\sqrt{\frac{1}{4\pi}}$ that encloses an isotropic point source at its centre.

When light is incident upon an interface, some of the flux is transmitted and some is reflected (neglecting absorption for the moment). The densities of transmitted and reflected flux relative to the incoming flux density are given by the Fresnel transmission and reflection formulae. According to electromagnetic theory, light travels in two independent types of waves called *polarizations*. Unpolarized light, which is usually the type of light considered in computer graphics, is an average of equal amplitudes of each of the two polarizations.

If $L_i\ \delta(\theta - \theta_i)$ is the radiance incident on some infinitesimal surface where θ_i is the angle between the surface normal and the direction of the incoming radiation, then the irradiance or flux density at the surface is given by $L_i \cos\theta_i$. Let $L_t\ \delta(\theta - \theta_t)$ be the transmitted radiance. Then $L_t \cos\theta_t$ is the transmitted flux density or exitance, and the ratio between the exitance and the irradiance, averaged over polarizations is given by the Fresnel transmission coefficient T [17]:

$$T(\theta_i) = 2\,\frac{n_t \cos\theta_t}{n_i \cos\theta_i}\left(\left[\frac{n_i \cos\theta_i}{n_i \cos\theta_i\ + n_t \cos\theta_t}\right]^2 + \left[\frac{n_i \cos\theta_i}{n_t \cos\theta_i\ + n_i \cos\theta_t}\right]^2\right). \tag{4.4}$$

Figure 4.2 shows $T(\theta_i)$ for the media used in this thesis, where $n_i = 1.0$ and $n_t = 1.1$.

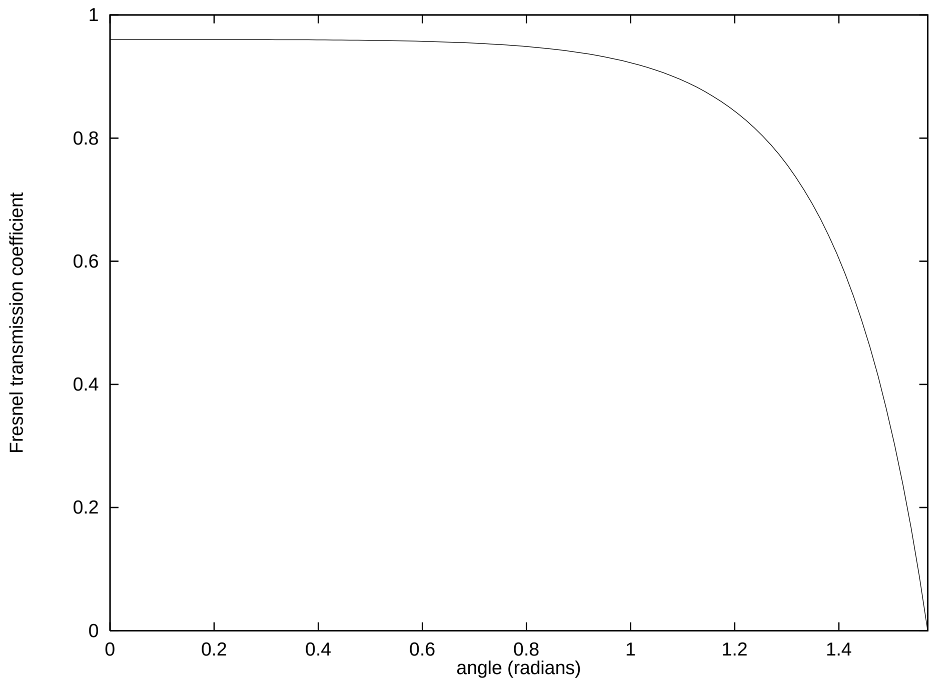

Figure 4.2: The Fresnel transmission coefficient as a function of incidence angle from the surface normal, for $n_i = 1.0$ and $n_t = 1.1$.

4.3 Modelling Water

The scattering of light in general is a difficult problem to solve. As light passes through some medium, it is repeatedly scattered by the matter that comprises the medium. The light emitted from each scattering can impinge upon matter in all other portions of the medium; thus each piece of matter in the medium can transmit and receive radiation from all other pieces of matter. This is called multiple scattering.

In order to model transmission of light through a medium, the geometric optics approximation neglects scattering at the atomic or molecular scale. In addition, the media are considered to be isotropic and homogeneous. Therefore, although the direction of the light deviates at a boundary between two media, the light travels in straight lines within each medium.

However, water may contain substantial particulate matter which causes a noticeable amount of scattering. In this work, multiple scattering is partially modelled by neglecting radiation which is *back scattered*, (light such that $\widehat{\mathbf{L_i}} \cdot \widehat{\mathbf{L_t}} < 0$), and modelling the multiple scattering of *forward scattered* ($\widehat{\mathbf{L_i}} \cdot \widehat{\mathbf{L_t}} > 0$) light by an (empirical) Gaussian distribution in angle about $\widehat{\mathbf{L_t}}$:

$$L(\ \widehat{\mathbf{P}}\ ) \sim \exp(-a(1 - (\widehat{\mathbf{L_t}} \cdot \widehat{\mathbf{P}}\ )^2)), \tag{4.5}$$

where $L(\widehat{\mathbf{P}})$ is the radiance in direction $\widehat{\mathbf{P}}$ and a is a coefficient proportional to the density of particulate matter, which indicates the diffusion of scattered light in the water. Although this is empirical, it is known that a sequence of constant–mean Gaussians with uniformly decreasing variances converges to the Dirac delta function [4], which would represent a light ray. This term is similar to a Gaussian in the sequence. It becomes more ray–like with increasing a, which corresponds with less particulate matter in the water. However, this term is not normalized and thus energy is not strictly conserved.

In addition, attenuation due to absorption is modelled by

$$L(\ d\ ) \sim \exp(-E\ d),$$

where $L(d)$ is the radiance at distance d travelled by the light and E is the *extinction coefficient*, which is dependent on the conductivity of the particulate matter. This type of extinction occurs when the absorption probability is constant with position: the light is exponentially attenuated with distance travelled [42].

4.4 Computing Caustics

Recall that a height field $h(\Omega)$ is defined on the domain $\Omega = [0, M\Delta x] \times [0, N\Delta y]$ and let $H = \max_{(x,y)\in\Omega} \mid h(x,y) \mid$. Define the surface of the water as

$$S = \{\ (x, y, h(x,y)) \ \mid\ (x,y) \in \Omega\ \}.$$

Let L_{source} be the radiance of a point source that is in the incident medium. Let D be the distance from the point source to the point $(0,0,0)$. Assume that the source is very distant: that is, that $M\Delta x$, $N\Delta y$, and H are all very much smaller than D. From these assumptions, we can consider $\widehat{\mathbf{L}}_{\mathbf{i}}$ to be constant on S.

Consider a differential area dS on the surface S, having unit surface normal $\widehat{\mathbf{N}}$. Then the irradiance on dS is

$$\int L_{source}\ \delta(\theta - \theta_i)\ \cos\theta_i\ d\omega \quad = \quad L_{source}\ \cos\theta_i,$$

where the integral is taken over directions θ above dS.

The irradiance at a plane below and parallel to the xy plane can be calculated in this model by integrating over the surface of the interface. If we assume that this plane is perfectly diffuse, and that light reflected from this plane is *not* absorbed by the transmission medium and not reflected from the surface boundary, then this is the image on the plane that would be seen by a viewer in the transmission medium.

Let $(x_0, y_0, -d)$ be some point on the plane at which the irradiance is to be calculated. Let $\vec{\mathbf{P}}$ be the vector from the point $(x, y, h(x, y))$ to the point $(x_0, y_0, -d)$. Let $\widehat{\mathbf{P}} = \vec{\mathbf{P}} \ / \ || \ \vec{\mathbf{P}} \ ||$. Then at the point $(x_0, y_0, -d)$ the irradiance is given by

$$L_{source} \int_{\Omega} T(\theta_i)\ \cos\theta_i\ (\widehat{\mathbf{P}}_z) \exp(-a(1 - (\widehat{\mathbf{L}}_{\mathbf{t}} \cdot \widehat{\mathbf{P}}\ )^2)) \exp(-E\ \ || \ \vec{\mathbf{P}} \ ||) dx\ dy, \tag{4.6}$$

where $\widehat{\mathbf{P}}_z = -\widehat{\mathbf{P}} \cdot (0, 0, -1)$.

The next chapter discusses how this expression for the irradiance is calculated at discrete points on the plane, using the graphics hardware. The irradiance yields an image of the caustic induced by the height field.

Chapter 5

Ray-tracing Refraction in Hardware

The calculations determine the light incident on a plane beneath an interface between two materials of differing indices of refraction. Incident light is limited to light of a single wavelength coming from a single direction. The effects of differential refraction and extinction by the transmission medium could be calculated using the same framework, but with a performance penalty as a number of the calculations would have to be repeated for each wavelength. In addition, the restriction to a single angle of incidence could be made less severe without incurring a large penalty by allowing a different angle of incidence at each sample point on the surface of the interface. However, allowing incident light on the surface to come from more than one direction at each sample point would be computationally expensive.

Due to the approximations described below, the current implementation is most accurate for incoming light directions which are close to $\widehat{\mathbf{L_i}} = (0, 0, -1)$, which corresponds to the midday sun position.

5.1 Approximation and Discretization

The problem to be solved is to integrate equation 4.6, which gives the irradiance on the plane $z = -d$. To do so, the integral must be discretized, and a number of approximations must be made in order to achieve real–time performance.

Let $\Omega = [0, W] \times [0, W]$, where $W = N\Delta x,\ N \in \mathbb{N}_0,\ \Delta x \in \mathbb{R}$.

Let $h\colon \Omega \to \mathbb{R}$ such that $|h(x, y)| \leq H\ \forall (x, y) \in \Omega$.

Let L_i be the radiance of the incident light in direction $\widehat{\mathbf{L_i}}$ (from the source)

Let L_t be the radiance of the refracted light in direction $\widehat{\mathbf{L_t}}$ after interacting with the surface. The index of refraction of the incident medium is considered to be n_i and that of the transmission medium is considered to be n_t.

Let the plane $z = -d$ at which the irradiance is being calculated be called the *floor plane*.

5.1.1 Discretization of the Height Field

For the discretized version, let a patch Ω_{ij} of the domain be

$$\Omega_{i,j} = [i\Delta x, (i+1)\Delta x] \times [j\Delta y, (j+1)\Delta y], (0 \leq i, j < N),$$

the corresponding patch on the height field be

$$A_{ij}(x, y) = h(\Omega_{i,j}),$$

and a corresponding point at the corner of a patch be

$$h_{ij} = h(i\Delta x, j\Delta y).$$

Let $\widetilde{\Omega} = \{(i, j) | i < N, j < M, i, j \in \mathbb{N}_0\}$. Let $\tilde{h}\colon \widetilde{\Omega} \to [0, 1]$ such that

$$\tilde{h}_{i,j} = \frac{h(i\Delta x, j\Delta y)}{2H} + 0.5.$$

Then $\tilde{h}_{i,j}$ is a discretization of the continuous height field h, and $\tilde{h}_{i,j} \in [0, 1]$.

In order to approximate the normal to the height field, we approximate each patch $A_{i,j}(x, y)$ by a portion of a plane. Then the partial derivatives of a patch are given by:

$$\frac{\partial A_{ij}}{\partial x} \approx \frac{\Delta_i h_{i,j}}{\Delta x} = \frac{h_{i+1,j} - h_{i,j}}{\Delta x} = \frac{2H\tilde{h}_{i+1,j} - 2H\tilde{h}_{i,j}}{\Delta x} = (\tilde{h}_{i+1,j} - \tilde{h}_{i,j}) \cdot \frac{2H}{\Delta x}$$

and

$$\frac{\partial A_{ij}}{\partial y} \approx \frac{\Delta_j h_{i,j}}{\Delta y} = \frac{h_{i,j+1} - h_{i,j}}{\Delta x} = \frac{2H\tilde{h}_{i,j+1} - 2H\tilde{h}_{i,j}}{\Delta x} = (\tilde{h}_{i,j+1} - \tilde{h}_{i,j}) \cdot \frac{2H}{\Delta x}.$$

The unit normal to a patch A_{ij} is then approximated by

$$\widehat{\mathbf{N}}_{ij} = \frac{\vec{N}_{ij}}{\|\vec{N}_{ij}\|} = \frac{(-2H\frac{\Delta_i \tilde{h}_{ij}}{\Delta x}, -2H\frac{\Delta_j \tilde{h}_{ij}}{\Delta x}, 1)}{\sqrt{(2H\frac{\Delta_i \tilde{h}_{ij}}{\Delta x})^2 + (2H\frac{\Delta_j \tilde{h}_{ij}}{\Delta x})^2 + 1}}. \tag{5.1}$$

5.1.2 Discretization of the Fresnel Transmission Coefficient

The light incident on a planar patch is considered to reflect and refract at the surface boundary. The surface is considered to be linear, isotropic, homogeneous, and non-conducting, while the light is considered to be unpolarized (that is, an equal mixture of both polarizations). In this situation, the fraction of the flux density that is transmitted can be modelled by the Fresnel transmission coefficient, $T(n_i, n_t, \cos\theta_i, \cos\theta_t)$ (see equation 4.4).

In the implementation, T is discretized as a function of $\cos\theta_i$ and evaluated with a lookup table. This is possible since $\cos\theta_t$ is also a function of $\cos\theta_i$ via Snell's law (equation 4.1) and the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$:

$$\begin{aligned}
& & n_i \sin\theta_i &= n_t \sin\theta_t \\
&\Rightarrow & n_i^2 \sin^2\theta_i &= n_t^2 \sin^2\theta_t \\
&\Leftrightarrow & n_i^2(1 - \cos^2\theta_i) &= n_t^2(1 - \cos^2\theta_t) \\
&\Leftrightarrow & \cos^2\theta_t &= 1 - \left(\tfrac{n_i}{n_t}\right)^2 (1 - \cos^2\theta_i).
\end{aligned}$$

In this situation, we are concerned with light for which $0 \leq \cos\theta_t \leq 1$, so the relationship between $\cos\theta_t$ and $\cos\theta_i$ becomes:

$$\cos\theta_t = \sqrt{1 - \left(\frac{n_i}{n_t}\right)^2 (1 - \cos^2\theta_i)}. \tag{5.2}$$

By composing equations 5.2 and 4.4, T can be expressed as a function of $\cos\theta_i$, n_i, and n_t only.

5.1.3 Discretization of Γ

In the same manner, equation 4.3 for Γ and equation 5.2 are composed so that Γ can be expressed as a function of $\cos\theta_i$, n_i, and n_t only. This function is also discretized and evaluated with a lookup table.

5.1.4 Discretization of the Integral

The integral 4.6 itself is approximated in order that we may trade computational exactness for computational efficiency. First of all, since the integral computes the irradiance at a point on the floor plane, we subdivide this plane into a set of patches $F_{ij} = [i\Delta x, (i+1)\Delta x] \times [j\Delta x, (j+1)\Delta x]$ where $i < N, j < M, i, j \in \mathbb{N}_0$, and consider the irradiance to be constant on each patch. Then the irradiance, E_{ij}, on a patch, F_{ij}, is given by the integral 4.6, where (x_0, y_0) is the point at the centre of a patch.

Even using the simplest quadrature schemes, it is impossible to compute N^2 such integrals in real-time with hardware currently commonly available. Therefore, the integrand must be carefully approximated such that computational efficiency is maximized without sacrificing too much exactness.

Since the imaging pipeline has the ability to perform convolutions, the integral can be approximated efficiently by phrasing it as a convolution and leveraging the efficiency of the hardware. However, in order to do so, the integrand must be *separable*, i.e. the integral must be able to be expressed as

$$\int f(x,y)\; g(x-x_0, y-y_0)\; dx\; dy.$$

Noting that a product of functions is separable if and only if each function in the product is separable, and recalling the terms in the integrand of integral 4.6, we have the following components:

$T(\theta_i)$:	Since $\theta_i = -\widehat{\mathbf{L}}_\mathbf{i} \cdot \widehat{\mathbf{N}}$ and $\widehat{\mathbf{N}}$ is a function of $\frac{\partial h(x,y)}{\partial x}$ and $\frac{\partial h(x,y)}{\partial y}$, $T = T(x,y)$ and this function is separable.
$\cos\theta_i$:	Similarly, $\theta_i = \theta_i(x,y)$, and this function is separable.
$\exp(-E \parallel \vec{\mathbf{P}} \parallel)$:	Since $\parallel \vec{\mathbf{P}} \parallel = \sqrt{(x_0-x)^2+(y_0-y)^2+(-d-h(x,y))^2}$, and $h(x,y)$ might not be separable, this function is not necessarily separable.
$\widehat{\mathbf{P}}_z$:	$\widehat{\mathbf{P}}_z = \frac{-d-h(x,y)}{\parallel \vec{\mathbf{P}} \parallel}$, and this function is not necessarily separable.
$\exp(-a(1-(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^2))$:	This function is not separable.

By making the approximation $d \gg h(x,y)$ so that $-d-h(x,y) \approx -d$, then $\exp(-E \parallel \vec{\mathbf{P}} \parallel)$ and $\widehat{\mathbf{P}}_z$ bbecome functions of (x_0-x, y_0-y) and are both separable.

Finally, an approximation is needed for the Gaussian, $\exp(-a(1-(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^2))$. To approximate this function, we use an exponentiated cosine: $(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^n$. This can be expanded as follows:

$$\begin{aligned}
(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^n &= (\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_x + \widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_y + \widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^n \\
&= \sum_{i=0}^{n}\sum_{j=0}^{n-i} \frac{n!}{i!j!(n-i-j)!}(\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_x)^i(\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_y)^j(\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-i-j}.
\end{aligned}$$

In order to simplify this, we make the approximation that $\widehat{\mathbf{L}}_{\mathbf{t}z} \gg \widehat{\mathbf{L}}_{\mathbf{t}x}, \widehat{\mathbf{L}}_{\mathbf{t}y}$. This is true when the direction of incidence is close to vertical and the slope of the waves is small.

Using this approximation, we keep terms to first order in $\widehat{\mathbf{L}}_{\mathbf{t}x}$ and $\widehat{\mathbf{L}}_{\mathbf{t}y}$, giving:

$$\begin{aligned}
(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^n &\approx (\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^n + n(\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_x)(\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-1} + n(\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_y)(\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-1} \\
&= (\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-1}(\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z + n(\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_x) + n(\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_y)).
\end{aligned}$$

Although this function is indeed separable, it suffers from the problem that it is not non–negative for the possible values of $\widehat{\mathbf{L}}_\mathbf{t}$ and $\widehat{\mathbf{P}}$ that will be encountered. This would result in irradiance begin subtracted from the sum for a given $(x_0, y_0, -d)$ when $\widehat{\mathbf{L}}_\mathbf{t}$ points far enough away from that position. Thus a further approximation is made by reducing the contribution of the terms containing $\widehat{\mathbf{L}}_{\mathbf{t}x}$ and $\widehat{\mathbf{L}}_{\mathbf{t}y}$:

$$
\begin{aligned}
(\widehat{\mathbf{L}}_{\mathbf{t}} \cdot \widehat{\mathbf{P}})^n &\approx (\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-1}(\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z + \widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_x + \widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_y) \\
&= (\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-1}(\widehat{\mathbf{L}}_{\mathbf{t}} \cdot \widehat{\mathbf{P}}) \\
&= \widehat{\mathbf{L}}_{\mathbf{t}z}^n\widehat{\mathbf{P}}_z^n + \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_z^{n-1}\widehat{\mathbf{P}}_x + \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_z^{n-1}\widehat{\mathbf{P}}_y.
\end{aligned}
$$

Since $\widehat{\mathbf{L}}_{\mathbf{t}}$ is a function of (x, y) and $\widehat{\mathbf{P}}$ is a function of $(x_0 - x, y_0 - y)$ under the approximation above, then this result is the sum of three separable functions.

Putting all these results together gives

$$
\begin{aligned}
L(x_0, y_0, -d) &\approx L_1 + L_2 + L_3 \\
&= \int_\Omega f_1(x,y)\; g_1(x_0 - x, y_0 - y)\; dx\, dy \\
&+ \int_\Omega f_2(x,y)\; g_2(x_0 - x, y_0 - y)\; dx\, dy \\
&+ \int_\Omega f_3(x,y)\; g_3(x_0 - x, y_0 - y)\; dx\, dy,
\end{aligned}
$$

where

$$
\begin{aligned}
f_1(x,y) &= T(x,y)\; \cos\theta_i\;(x,y)\; \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}\widehat{\mathbf{L}}_{\mathbf{t}x}, \\
f_2(x,y) &= T(x,y)\; \cos\theta_i\;(x,y)\; \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}\widehat{\mathbf{L}}_{\mathbf{t}y}, \\
f_3(x,y) &= T(x,y)\; \cos\theta_i\;(x,y)\; \widehat{\mathbf{L}}_{\mathbf{t}z}^{n}, \\
g_1(x_0 - x, y_0 - y) &= \widehat{\mathbf{P}}_z^n\widehat{\mathbf{P}}_x\; \exp(-E\;\; ||\,\vec{\mathbf{P}}\,||), \\
g_2(x_0 - x, y_0 - y) &= \widehat{\mathbf{P}}_z^n\widehat{\mathbf{P}}_y\; \exp(-E\;\; ||\,\vec{\mathbf{P}}\,||), \\
g_3(x_0 - x, y_0 - y) &= \widehat{\mathbf{P}}_z^{n+1}\; \exp(-E\;\; ||\,\vec{\mathbf{P}}\,||).
\end{aligned}
$$

Thus the L becomes a sum of three convolutions where the functions g_m are the kernels.

Discretizing the continuous integrals gives

$$\begin{aligned}
L_1 &= L_s \sum\sum \left[T \ \cos\theta_t \ \ (\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1} \ \widehat{\mathbf{L}}_{\mathbf{t}x}\right] \left[(\widehat{\mathbf{P}}_z)^n \ \widehat{\mathbf{P}}_x \ \exp(-E\|\vec{P}\|)\right], \\
L_2 &= L_s \sum\sum \left[T \ \cos\theta_t \ \ (\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1} \ \widehat{\mathbf{L}}_{\mathbf{t}y}\right] \left[(\widehat{\mathbf{P}}_z)^n \ \widehat{\mathbf{P}}_y \ \exp(-E\|\vec{P}\|)\right], \\
L_3 &= L_s \sum\sum \left[T \ \cos\theta_t \ \ (\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1} \ \widehat{\mathbf{L}}_{\mathbf{t}z}\right] \left[(\widehat{\mathbf{P}}_z)^{n+1} \ \exp(-E\|\vec{P}\|)\right],
\end{aligned}$$

where L_s is the source radiance, and the double sum is over the size of the kernel.

5.2 Algorithm

The algorithm is as follows:

INPUT:

$\tilde{h}_{ij}$	Discrete height field, scaled and biased into $[0, 1]$
W	Length of side of square domain on which h is defined
H	Absolute bound on the magnitude of h
n_i	Index of refraction of incident medium
n_t	Index of refraction of transmission medium
d	Depth of plane on which irradiance of refracted light is calculated
E	Extinction coefficient
a	Diffusion coefficient
L_s	Light source radiance
$\widehat{\mathbf{L}}_{\mathbf{t}}$	Direction of incoming radiance from source

OUTPUT:

E	Irradiance on plane $z = -d$

BEGIN:

Stage 0: Calculate lookup tables, convolution kernels

FOR EACH HEIGHT FIELD IN THE SEQUENCE OF TIME–VARYING HEIGHT FIELDS

Stage 1: Calculate $\widehat{\mathbf{N}}_{ij}$, vector field of normals to height field

Stage 2: Calculate $\Gamma\ \widehat{\mathbf{N}}_{ij}$, term needed to determine $\widehat{\mathbf{L}}_{\mathbf{i}}$

Stage 3: Calculate $I_t = L_s \cos\theta_i\ T$

Stage 4: Calculate $\widehat{\mathbf{L}}_{\mathbf{t}}$

Stage 5: Calculate function to be convolved

Stage 6: Convolve function using kernel

REPEAT

END

5.2.1 Stage 0: Calculation of Tables and Kernels

In addition to a time–dependent height field, all input variables can be interactively modified. On modification, the appropriate lookup tables and kernels are recalculated, and the rendering of the refraction continues in the new state. The following is a summary of actions taken on the change of inputs:

H Changing the vertical scale factor of the height field necessitates changing an internal variable which controls the scaling of derivatives.

W Changing the horizontal scale factor of the height field requires changing both the internal variable which controls the scaling of derivatives, and recomputing the filters which are used for the final convolution.

n_i, n_t Changing the indices of refraction necessitates recalculation of the relative index of refraction and the Fresnel transmission and Γ lookup tables.

d Changing the depth of the floor plane requires recalculating the filters used for the final convolution.

E Changing the extinction factor requires recalculating the filters used for the

HOST MEMORY FRAMEBUFFER

$$
\begin{bmatrix} \tilde{h}_{i,j} \end{bmatrix}
\underset{\substack{\text{L}\\ \downarrow \\ \text{RGB}}}{\longrightarrow}
\begin{bmatrix} \tilde{h} \\ \tilde{h} \\ \tilde{h} \\ 1. \end{bmatrix}
\underset{\text{CONV}}{\longrightarrow}
\begin{bmatrix} \Delta_i \tilde{h} \\ \Delta_j \tilde{h} \\ 0. \\ 1. \end{bmatrix}
\underset{\text{SB2}}{\longrightarrow}
\begin{bmatrix} \frac{\Delta_i h}{\Delta x} \\ \frac{\Delta_j h}{\Delta y} \\ 0. \\ 1. \end{bmatrix}
\underset{\substack{\text{PIXEL}\\ \text{TEXTURE}}}{\longrightarrow}
\begin{bmatrix} -\alpha \hat{N}_x + \beta \\ -\alpha \hat{N}_y + \beta \\ -\alpha \hat{N}_z + \beta \\ 1. \end{bmatrix}
$$

Figure 5.1: Stage 1: Calculation of height field normals. Subscripts have been omitted for clarity.

final convolution.

a Changing the diffusion coefficient necessitates changing an internal variable which indexes the lookup table to be used for exponentiation.

L_s Changing the light source radiance requires changing a variable used to scale the result of the final convolution.

$\widehat{\mathbf{L}}_{\mathbf{t}}$ Changing the direction of incidence necessitates changing the values of the internal variables which store the values for the vector.

5.2.2 Stage 1: Calculation of Height Field Normals

Figure 5.1 shows the flow of data through the graphics pipeline during the computation of heightfield normals. The values of the height field $\tilde{h}_{ij}$ that are initially drawn to the framebuffer must be in the interval $[0, 1]$ in order to avoid clamping. They are drawn as LUMINANCE values, and the L $\rightarrow$ RGB operation copies the height into each of the R,G, and B components. The A component is set to 1.0.

The convolution operation calculates the discrete derivative. Different filters are define for each of the R,G, and B planes which allow the discrete derivative in both the i and j directions to be taken in the same pass.

Since, h_{ij} may take on arbitrary values as long as it is bounded on the domain of definition, the scale factors W and H corresponding to the physical situation are supplied. This compensates for the fact that $\tilde{h}_{ij}$ is in $[0, 1]$, and allows the discrete derivative of the original function h_{ij} to be calculated. This operation SB2 performs not only this scaling, but it also provides the scale and bias necessary to ensure that the values in the pixel rectangle at this point in the pipeline are in the interval $[0, 1]$ as the next operation (PIXEL TEXTURE) requires.

The R and B components are then used to look up the normalized vector values using the PIXEL TEXTURE operation. These normalized values have been pre–computed, scaled and biased, and stored in the R,G, and B components of a texture map. The scale and bias is necessary since texture map values must be in the interval $[0, 1]$.

Due to ease of manipulation, the normalized vector that is looked up is actually $-\widehat{\mathbf{N}}$, where the sense of $\widehat{\mathbf{N}}$ is defined in Figure 4.1. This makes some of the operations such as calculating $\cos\theta_i$ a little easier later.

At the end of Stage 1, the scaled and biased normals to the height field are stored in the framebuffer.

5.2.3 Stage 2: Calculation of $\Gamma\ \widehat{\mathbf{N}}$

This stage requires two passes, shown in Figure 5.2. The first pass is to prepare for the eventual blend at the end of the second pass, and is done in order to ensure the correct biasing of $-\Gamma\ \widehat{\mathbf{N}}_{ij}$ by the end of this stage. It consists of drawing the constant bias value β to the framebuffer as a LUMINANCE value. This value is then copied into the R,G, and B components during the L $\rightarrow$ RGB operation and copied into the framebuffer. This pass is not very computationally expensive as all other operations have been disabled.

The second pass consists of copying the scaled and biased normal values that are in the framebuffer as a result of Stage 1 to another position in the framebuffer. This has the effect of putting the pixel rectangle through the imaging pipeline. The first operation, SB1 serves to un–scale and un–bias the normals, after which the CM operation sums

HOST MEMORY $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ FRAMEBUFFER

$$\begin{bmatrix} \beta \end{bmatrix} \;\; \xrightarrow[\substack{\downarrow \\ \mathrm{RGB}}]{\mathrm{L}} \;\; \begin{bmatrix} \beta \\ \beta \\ \beta \\ 1. \end{bmatrix} \xrightarrow{\qquad\qquad\qquad\qquad\qquad} \begin{bmatrix} \beta \\ \beta \\ \beta \\ 1. \end{bmatrix}$$

FRAMEBUFFER
(From Stage #1)

$$\begin{bmatrix} -\alpha\widehat{N}_x+\beta \\ -\alpha\widehat{N}_y+\beta \\ -\alpha\widehat{N}_z+\beta \\ 1. \end{bmatrix} \xrightarrow{\mathrm{SB}1} \begin{bmatrix} -\widehat{N}_x \\ -\widehat{N}_y \\ -\widehat{N}_z \\ 1. \end{bmatrix} \underset{\|}{\xrightarrow{\mathrm{CM}}} \begin{bmatrix} -\widehat{N}_x \\ -\widehat{N}_y \\ -\widehat{N}_z \\ -\widehat{\mathbf{L}}_{\mathbf{i}} \cdot \widehat{N} \end{bmatrix} \xrightarrow{\mathrm{SB}2} \begin{bmatrix} -\alpha\widehat{N}_x+\beta \\ -\alpha\widehat{N}_y+\beta \\ -\alpha\widehat{N}_z+\beta \\ -\cos\theta_i \end{bmatrix} \xrightarrow{\mathrm{CL}3} \begin{bmatrix} -\alpha\widehat{N}_x+\beta \\ -\alpha\widehat{N}_y+\beta \\ -\alpha\widehat{N}_z+\beta \\ -\Gamma(\cos\theta_i) \end{bmatrix} \bullet (-\Gamma)$$

$$\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 0. \\ \widehat{\mathbf{L}}_{\mathbf{i}x} & \widehat{\mathbf{L}}_{\mathbf{i}y} & \widehat{\mathbf{L}}_{\mathbf{i}z} & 0. \end{bmatrix}$$

$$+$$

$$\text{BLEND} \downarrow \quad \begin{bmatrix} \beta \\ \beta \\ \beta \\ 1. \end{bmatrix} \bullet \;(1+\Gamma)$$

$$\text{FRAMEBUFFER} \quad \begin{bmatrix} \alpha\Gamma\widehat{N}_x+\beta \\ \alpha\Gamma\widehat{N}_y+\beta \\ \alpha\Gamma\widehat{N}_z+\beta \\ \Gamma^2-\Gamma+1. \end{bmatrix}$$

Figure 5.2: Stage 2: Calculation of $\Gamma\ \widehat{\mathbf{N}}$. $\Gamma(\cos\theta_i)$ is a scalar function defined on the discrete set Ω. A constant valued ($=\beta$) scalar function also defined on Ω is first drawn to the framebuffer, so that proper biasing is maintained at the end of the stage.

a linear combination of them. Since the normalized incident direction vector has been previously placed in the colour matrix, this operation takes the dot product of $\widehat{\mathbf{L}}_{\mathbf{i}}$ and $-\widehat{\mathbf{N}}$, which is $\cos\theta_i$. The CM is such that the dot product is placed in the ALPHA component and the normal values are preserved.

Application of lookup tables causes clamping of values to $[0, 1]$. Since a lookup table will be applied later in the pipeline, the normal values must be re–scaled and re–biased into $[0, 1]$, which is accomplished by the SB2 operation. However, the value in the ALPHA position is now $\cos\theta_i$, and it is neither scaled nor biased.

The next operation is the colour lookup in order to determine Γ as a function of $\cos\theta_i$. This function has been pre–computed, and is in the interval $[0, 1]$ so needs no biasing. The Γ value is placed in the ALPHA component, replacing $\cos\theta_i$.

At this point in the pipeline the R,G, and B components of the pixel rectangle contain the scaled and biased components of the normal, while the ALPHA component contains Γ. In order to calculate $-\Gamma\,\widehat{\mathbf{N}}$, the BLEND operation is invoked. The source blend factor is the ALPHA component of the source, which is Γ, while the destination blend factor is $(1 - \Gamma)$. The destination is the constant valued pixel rectangle from the first pass of this stage, which serves to leave the product $-\Gamma\,\widehat{\mathbf{N}}$ properly scaled and biased at the end of the blend.

5.2.4 Stage 3: Calculation of $I_t = L_s \cos\theta_i \; T$

This stage also consists of two passes, shown in Figure 5.3. The first pass copies the incident radiance values into the framebuffer from host memory. They will be used in the BLEND operation at the end of the second pass to calculate the product $L_s\; T$. The values are copied in LUMINANCE mode, and are thus copied into each of the R,G, and B components.

The second pass copies the scaled and biased normals which were left in the framebuffer at the end of Stage 1 to another region of the framebuffer. As in the previous pass,

HOST MEMORY → FRAMEBUFFER

$$\begin{bmatrix} L_s \end{bmatrix} \underset{\text{RGB}}{\overset{\text{L}}{\rightarrow}} \begin{bmatrix} L_s \\ L_s \\ L_s \\ 1. \end{bmatrix} \longrightarrow \begin{bmatrix} L_s \\ L_s \\ L_s \\ 1. \end{bmatrix}$$

FRAMEBUFFER (From Stage #1)

$$\begin{bmatrix} -\alpha\widehat{N}_x + \beta \\ -\alpha\widehat{N}_y + \beta \\ -\alpha\widehat{N}_z + \beta \\ 1. \end{bmatrix} \overset{\rightarrow}{\text{SB}1} \begin{bmatrix} -\widehat{N}_x \\ -\widehat{N}_y \\ -\widehat{N}_z \\ 1. \end{bmatrix} \overset{\rightarrow}{\text{CM}} \begin{bmatrix} -\widehat{N}_x \\ -\widehat{N}_y \\ -\widehat{N}_z \\ -\widehat{\mathbf{L}}_{\mathbf{i}} \cdot \widehat{N} \end{bmatrix} \overset{\rightarrow}{\text{CL}3} \begin{bmatrix} -\widehat{N}_x \\ -\widehat{N}_y \\ -\widehat{N}_z \\ \widetilde{T}(\cos\theta_i\ ) \end{bmatrix} \bullet\ 0$$

$$\text{CM} = \begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 0. \\ \widehat{\mathbf{L}}_{\mathbf{i}x} & \widehat{\mathbf{L}}_{\mathbf{i}y} & \widehat{\mathbf{L}}_{\mathbf{i}z} & 0. \end{bmatrix}$$

$$+ \begin{bmatrix} L_s \\ L_s \\ L_s \\ 1. \end{bmatrix} \bullet\ \widetilde{T}$$

BLEND ↓ FRAMEBUFFER

$$\begin{bmatrix} I_t \\ I_t \\ I_t \\ \widetilde{T} \end{bmatrix}$$

Figure 5.3: Stage 3: Calculation of transmitted irradiance (exitance) $I_t = L_s\widetilde{T} = L_s \cos\theta_i\ T$. L_s and I_t are scalar functions defined on the discrete set $\tilde{\Omega}$; subscripts have been omitted for clarity. L_s is first drawn to the framebuffer in order to maintain proper biasing at the end of the stage.

operation SB1 serves to un–scale and un–bias the normals.

The same colour matrix is applied as in Stage 2, leaving $\cos\theta_i$ in the ALPHA component. However, in this stage the normals are not re–scaled and re–biased as they will not be needed again in this pass.

The next operation is CL3. This is used to look up the Fresnel transmission coefficient, T, which is left in the ALPHA component.

Finally, the BLEND operation is invoked to calculate the product $L_s \cos\theta_i\ T$. Since both these functions were previously in the interval $[0, 1]$, the product needs to be neither scaled nor biased.

5.2.5 Stage 4: Calculation of $\widehat{\mathbf{L_t}}$

In order to facilitate correct biasing at the end of the second pass of Stage 4, the constant valued pixel rectangle with value β is drawn to the framebuffer in a first pass, as shown in Figure 5.4.

In the second pass, the vector field $-\Gamma\ \widehat{\mathbf{N}}$ from Stage 2 is copied from one area of the framebuffer to another. The SB1 operation is used to add the constant vector

$$\alpha\ \frac{n_i}{n_t}\ \widehat{\mathbf{L_i}}$$

to the normals where α is the factor by which the normals are scaled, giving

$$\alpha\ (-\Gamma\ \widehat{\mathbf{N}}\ + \frac{n_i}{n_t}\ \widehat{\mathbf{L_i}}\ ) + \beta,$$

which is $\widehat{\mathbf{L_t}}$. (see equation 4.2)

The colour matrix is then used to copy the scaled and biased z component of $\widehat{\mathbf{L_t}}$ into the ALPHA position with operation CM.

The CL3 operation is then used to look up $\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}$, which replaces $\widehat{\mathbf{L}}_{\mathbf{t}z}$ in the ALPHA component.

The final operation of this pass is the BLEND operation. The source blend factor is

HOST MEMORY

FRAMEBUFFER

$$\begin{bmatrix}\beta\end{bmatrix} \quad \begin{matrix}\rightarrow\\ \text{L}\\ \downarrow\\ \text{RGB}\end{matrix} \quad \begin{bmatrix}\beta\\ \beta\\ \beta\\ 1.\end{bmatrix} \xrightarrow{\qquad\qquad\qquad\qquad} \begin{bmatrix}\beta\\ \beta\\ \beta\\ 1.\end{bmatrix}$$

FRAMEBUFFER
(From Stage #2)

$$\begin{bmatrix}\alpha\Gamma\widehat{N}_x+\beta\\ \alpha\Gamma\widehat{N}_y+\beta\\ \alpha\Gamma\widehat{N}_z+\beta\\ 1.\end{bmatrix} \underset{\text{SB1}}{\rightarrow} \begin{bmatrix}\alpha(\Gamma\widehat{N}_x+\frac{n_1}{n_2}\widehat{\mathbf{L}}_{\mathbf{i}x})+\beta\\ \alpha(\Gamma\widehat{N}_y+\frac{n_1}{n_2}\widehat{\mathbf{L}}_{\mathbf{i}y})+\beta\\ \alpha(\Gamma\widehat{N}_z+\frac{n_1}{n_2}\widehat{\mathbf{L}}_{\mathbf{i}z})+\beta\\ 1.\end{bmatrix} \underset{\text{CM}}{\rightarrow} \begin{bmatrix}\alpha\widehat{\mathbf{L}}_{\mathbf{t}x}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}y}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}z}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}z}+\beta\end{bmatrix} \underset{\text{CL3}}{\rightarrow} \begin{bmatrix}\alpha\widehat{\mathbf{L}}_{\mathbf{t}x}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}y}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}z}+\beta\\ \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}\end{bmatrix} \bullet\ \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}$$

$$\text{CM} = \begin{bmatrix}1. & 0. & 0. & 0.\\ 0. & 1. & 0. & 0.\\ 0. & 0. & 1. & 0.\\ 0. & 0. & 1. & 0.\end{bmatrix}$$

$$+\ \begin{bmatrix}\beta\\ \beta\\ \beta\\ 1.\end{bmatrix} \bullet\ (1-\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1})$$

$$\Big\downarrow\ \text{BLEND}$$

FRAMEBUFFER

$$\begin{bmatrix}\alpha\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta\\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta\\ \widehat{\mathbf{L}}_{\mathbf{t}z}^{2n-2}+1-\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}\end{bmatrix}$$

Figure 5.4: Stage 4: Calculation of $\widehat{\mathbf{L}}_{\mathbf{t}}$. A constant valued (=β) scalar function is first drawn to the framebuffer, so that proper biasing is maintained at the end of the stage.

$\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}$ (source ALPHA) and the destination blend factor is $1 - \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}$ (1 − source ALPHA). This has the effect of multiplying the R,G, and B components by the ALPHA component.

The resulting vector field is the properly scaled and biased function which will undergo one more pass to prepare it for the final convolution.

5.2.6 Stage 5: Preparation for Final Convolution

This stage, shown in Figure 5.5, completes the calculation of the function to be convolved in the final stage by multiplying the result of Stage 3 (I_t) by the result of Stage 4.

The first operation as I_t moves through the pipeline is the CM operation. This copies I_t into the ALPHA component, and zeros the R,G, and B components.

The SB3 operation then biases the R,G, and B components by β in preparation for proper biasing after the blend.

Finally, the BLEND operation multiplies the destination R,G, and B components by the source ALPHA component. This is achieved with correct biasing by specifying a source blend factor of 1 − source ALPHA $(1 - I_t)$ and a destination blend factor of ALPHA (I_t).

The resulting vector function is the function to be convolved; it has no dependence on the position of incident light on the plane below the interface.

5.2.7 Stage 6: Final Convolution

The final stage, shown in Figure 5.6, is the convolution which sums the contribution of flux density from the interface to give the irradiance on the plane below the interface.

The first operation in the pipeline is the SB1 operation which un-scales and un-biases the function calculated in Stage 5.

The result is then convolved using the CONV operation with a pre-calculated kernel, and after summing the R,G, and B components, the result is the irradiance on the plane below the interface.

FRAMEBUFFER
(From Stage #4)

$$\begin{bmatrix} \alpha\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \widehat{\mathbf{L}}_{\mathbf{t}z}^{2n-2}+1-\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} \end{bmatrix}$$

FRAMEBUFFER
(From Stage #3)

$$\begin{bmatrix} I_t \\ I_t \\ I_t \\ \tilde{T} \end{bmatrix} \xrightarrow[\text{CM}]{} \begin{bmatrix} 0. \\ 0. \\ 0. \\ I_t \end{bmatrix} \xrightarrow[\text{SB}3]{} \begin{bmatrix} \beta \\ \beta \\ \beta \\ I_t \end{bmatrix} \bullet (1-I_t)$$

$$\text{CM} = \begin{bmatrix} 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. \\ 1. & 0. & 0. & 0. \end{bmatrix}$$

$$+$$

$$\begin{bmatrix} \alpha\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \alpha\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \widehat{\mathbf{L}}_{\mathbf{t}z}^{2n-2}+1-\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} \end{bmatrix} \bullet I_t$$

BLEND

$$\downarrow$$

FRAMEBUFFER

$$\begin{bmatrix} \alpha I_t\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \alpha I_t\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ \alpha I_t\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1}+\beta \\ A \end{bmatrix}$$

Figure 5.5: Stage 5: Preparation for Convolution. The result of Stage 3 is processed and blended with the result of Stage 4 in order to calculate the function necessary for convolving in Stage 6.

$$\begin{array}{ccccccc}
\begin{array}{c}\textsc{Framebuffer}\\ \text{(From Stage \#5)}\end{array} & & & & & & \textsc{Framebuffer}\\
\begin{bmatrix} \alpha I_t \widehat{\mathbf{L}}_{\mathbf{t}x} \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} + \beta \\ \alpha I_t \widehat{\mathbf{L}}_{\mathbf{t}y} \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} + \beta \\ \alpha I_t \widehat{\mathbf{L}}_{\mathbf{t}z} \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} + \beta \\ A \end{bmatrix} & \begin{array}{c}\rightarrow\\ \textsc{sb}1\end{array} & \begin{bmatrix} I_t \widehat{\mathbf{L}}_{\mathbf{t}x} \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} \\ I_t \widehat{\mathbf{L}}_{\mathbf{t}y} \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} \\ I_t \widehat{\mathbf{L}}_{\mathbf{t}z} \widehat{\mathbf{L}}_{\mathbf{t}z}^{n-1} \\ A \end{bmatrix} & \begin{array}{c}\rightarrow\\ \textsc{conv}\\ \|\end{array} & \begin{bmatrix} E_{fx} \\ E_{fy} \\ E_{fz} \\ 0. \end{bmatrix} & \begin{array}{c}\rightarrow\\ \textsc{cm}\\ \|\end{array} & \begin{bmatrix} E_f \\ E_f \\ E_f \\ 0. \end{bmatrix}
\end{array}$$

$$\begin{bmatrix} \widehat{P}_x \widehat{P}_z^n \exp(-E\|\vec{P}\|) \\ \widehat{P}_y \widehat{P}_z^n \exp(-E\|\vec{P}\|) \\ \widehat{P}_z \widehat{P}_z^n \exp(-E\|\vec{P}\|) \end{bmatrix} \begin{bmatrix} 1. & 1. & 1. & 0. \\ 1. & 1. & 1. & 0. \\ 1. & 1. & 1. & 0. \\ 0. & 0. & 0. & 0. \end{bmatrix}$$

Figure 5.6: Stage 6. Here $E_{fk} = \sum\sum I_t (\widehat{\mathbf{L}}_{\mathbf{t}z} \widehat{P}_z)^{n-1} \widehat{P}_z \widehat{P}_k \widehat{\mathbf{L}}_{\mathbf{t}k} \exp(-E\|\vec{P}\|)$ and $E_f = \sum\sum I_t (\widehat{\mathbf{L}}_{\mathbf{t}z} \widehat{P}_z)^{n-1} \widehat{P}_z (\widehat{P} \cdot \widehat{\mathbf{L}}_{\mathbf{t}}\ ) \exp(-E\|\vec{P}\|)$

Chapter 6

Results and Conclusions

The algorithm was implemented in OpenGL on an SGI Indigo 2 and on an SGI Infinite Reality. The Infinite Reality does not support pixel texturing and, despite the documentation to the contrary, the author was unable enable pixel texturing on the Indigo 2. The pixel texturing was thus simulated in software. A separate, completely software implementation was also made.

The algorithms were tested on sequences of height fields generated by a separate program (which also used graphics hardware to generate the height fields, by the way). The animated height fields were of several circular waves spreading in a square pond, as from a set of dropped pebbles. The waves interfered with each other and reflected of the walls of the pond, making for a complex height field. One frame of the animated heightfield is shown in Figure 6.1.

Table 6.1 reports the frame rates for three configurations:

- The hardware implementation without pixel textures, which gives incorrect images but has times closest to those likely if pixel texturing were indeed supported.

- The hardware implementation with software pixel textures, which provides a lower bound on the frame rate.

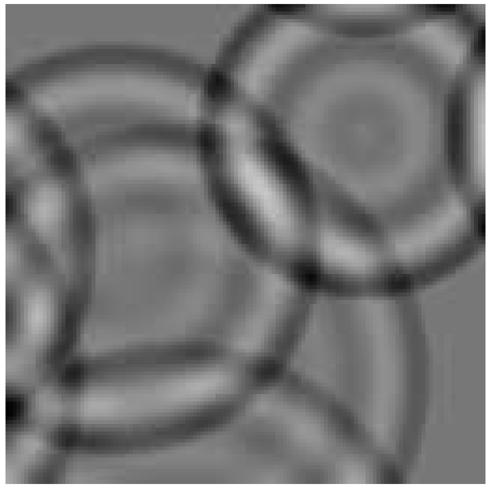

Figure 6.1: One frame of the animated height field. Bright regions are high, while dark regions are low.

- The software implementation.

From Table 6.1, it is clear that interactive rates of between 10 and 14 frames per second can be achieved on the Infinite Reality for a height field of size 256×256. On the Indigo 2, the frame rate drops to between three and four frames per second which, while not interactive, is still reasonable. The software implementation is between 8 and 26 times slower than the hardware implementation.

One concern is that this might not be a fair comparison between the software and hardware since the software has not been sufficiently optimized. However, this is belied by the substantial decrease in frame rate (of about 30%) when pixel textures are implemented in software. Since software pixel textures were strongly optimized, one would expect that shifting even more of the computation to software would further reduce the frame rate, to far below that achievable by the hardware implementation. In addition, the hardware version was not optimized, although one would expect less speedup from doing so as there are fewer possible optimizations.

A sequence of images from an animated height field is shown in Figure 6.2. Note

Table 6.1: Frames per second for various configurations of the algorithm on height fields of size 128×128 and 256×256, using a 7×7 convolution kernel.

	128×128	256×256
hardware (no pixel textures)		
Infinite Reality	37.94	14.86
Indigo 2	11.89	4.22
hardware (software pixel textures)		
Infinite Reality	24.84	9.38
Indigo 2	8.55	2.58
software		
Infinite Reality	2.56	0.56
Indigo 2	1.00	0.20

the decrese in height of the height field, characterized by a decrease in contrast, and the corresponding decrease in contrast of the refracted image. This corresponds to reduced refraction as the height of the waves decreases.

The implementation allows the user to change the light position and many of the parameters interactively. Figure 6.3 shows the caustics produced from the height field of Figure 6.1 at various angles of the light source. When the incident angle changes, so do the positions of the refracted rays, as can be seen. The exitance from the surface also changes since the transmitted irradiance is angle dependent, and therefore increasing angles of incidence yield darker images.

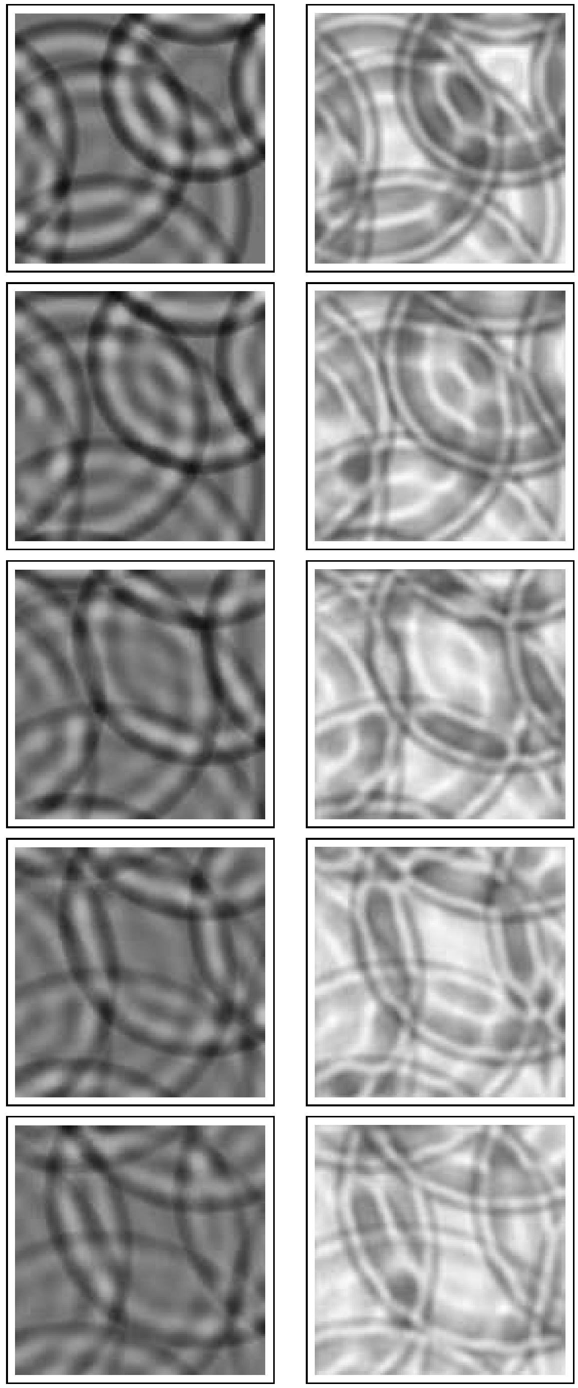

Figure 6.2: The caustics produced by the hardware implementation for a complex wave form made from the sum of circular ripples and their reflections. The light source in the (vertical) $y = 0$ plane at an angle of 0 degrees from vertical. The parameters are: diffusion coefficient $a = 14.0$, extinction factor $E = 0.04$, pond width 50 units, wave height 13.2 units, caustic plane depth 27.8 units, and index of refraction of water 1.3. Every tenth frame is shown.

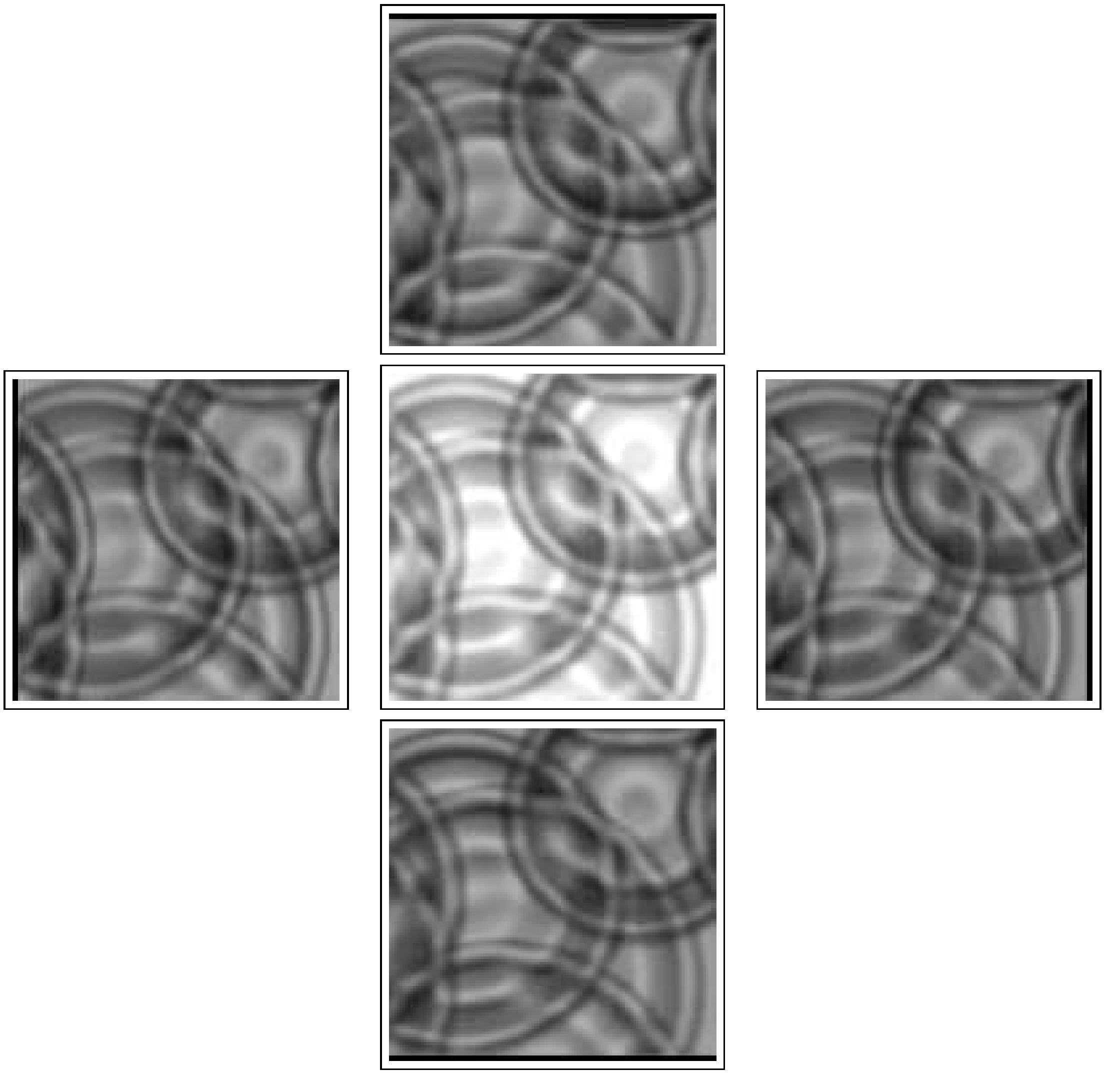

Figure 6.3: The caustics produced by the hardware implementation, for an incident light vector in the (vertical) $x = 0$ plane at angles of +10 (top), -10 (bottom) degrees from vertical and in the (horizontal) $y = 0$ plane at angles of +10 (right) and -10 (left) degrees from vertical. The incident light vector for the centre image is directly overhead, along the line of intersection of the $x = 0$ and $y = 0$ planes. The parameters are the same as those in Fig. 6.2. Note the displacement of the images; the black areas correspond to areas which would be in shadow due to the sides of square vessel containing the water.

6.1 Discussion

The images of Figure 6.2 represent a deterministic calculation of light refracted from a time–varying surface. This kind of calculation is very important when one can see both the surface and the plane on which the caustics are visible as the eye would easily perceive a mismatch otherwise. Situations in which this is true include shallow water, cut-away views, and viewpoints which are looking up from under the water, where the plane on which the caustics are visible is the image plane.

Although the conditions for the validity of approximations stated in Chapter 5 were respected in order to create the images shown above, the visual results seem reasonable for a larger range of parameters. This is because the approximation $d \gg h(x, y)$ is actually more restrictive than need be for a visual approximation — this condition doesn't influence the image much. This approximation was made for two reasons : so that the function for the extinction would be separable and so that the $\cos\phi$ — the cosine of the angle of incoming light with the floor plane normal — would be separable. However, at small floor plane depths, there is not much extinction and this term is not so important. In addition, $\cos\phi$ is not changed dramatically enough by the approximation to disturb the general functional form of the caustics.

The convolution is taken over the surface patch which is centred on the intersection of the height field and the line in the direction of the incoming light which passes through the point in question. Thus the convolution kernel is directly overhead when the light source is directly overhead, and displaced in the direction of the source when the incident light angle changes. This captures the most important contribution from the refracted light to the point on the floor plane.

The images are monochromatic, but a full colour image could quickly be approximated, or could be explicitly calculated using the wavelength dependent Fresnel coefficient, index of refraction, and extinction coefficient. This latter approach would take up to three times the computational effort, however.

It is clear from the images that the general features of refraction from a time-dependent surface have been calculated. However what is not clear from the images is that the calculation involves a multi–step process in which geometric optics is used to calculate the image of the caustics with the aid of some approximation. Since this is the crux of this thesis it is emphasized here: *Complex mathematical calculations can be performed in graphics hardware in real–time.*

6.2 Conclusions

The work presented here shows that graphics hardware can be used for calculations other than those for which it was designed. Refractive caustics were calculated at interactive rates and although the calculations were image–related, many of the calculations involved more abstract geometry and physics.

Hardware Improvements

This opens the door to the possibility of doing many more types of calculations in the graphics hardware. Several improvements in the hardware could be made to facilitate this, such as:

- greater precision in buffers
- signed arithmetic
- expanded convolutions
- pixel texturing
- improved communication between geometry and image processing

Deeper buffers — providing more bits of precision — would expand the type and number of calculations possible since those calculations which require greater numerical accuracy would be achievable. It is interesting to note that the accuracy of the result of calculations need only be as good as that of the human eye; depending on the stability of the calculation, one often may not need the precision that is seen these days in central processors.

Signed arithmetic would be a welcome addition to the pipeline. As seen in Chapter 3, the calculations necessary to support signed arithmetic and avoid clamping can become quite involved. This makes the hardware difficult to use for this purpose.

Convolutions are a fundamental operation in mathematics, particularly in signal processing. It is unfortunate that kernel sizes are limited, and any expansion of this limit would provide more power to the programmer.

Pixel texturing is an experimental extension provided on some machines by SGI. It is a very powerful feature, and would be an important addition to hardware capable of performing more general calculations.

In the same line of thought as pixel texturing, it would be useful to be able to interpret pixel R,G, and B values as spatial coordinates of vertices and vice-versa. This would give full access to/from the imaging pipeline from/to the geometry pipeline, and would greatly enhance the computational power.

A Basis Set of Operations

Since the display of images involves complicated mathematical calculations, it makes sense to analyse these calculations and to try to derive a "basis set" of mathematical operations with which these calculations can be performed in real–time.

Looking beyond computer graphics, one could determine which calculations from other domains are also amenable to hardware–assisted computation. Such calculations could include compute–intensive parallel problems in physics and computer vision.

However, a calculation is amenable to hardware–assisted computation only if it does not require a high degree of precision in the intermediate and final stages of the calculations, and only if suitable approximations and discretizations of the components of the calculation exist to permit hardware implementation.

It could be quite rewarding to survey other fields, find these calculations that are amenable to hardware–assistance, and to develop the basis set of graphics hardware operations that allows one to perform these calculations in hardware.

Automatic Assistance

Using the existing imaging pipeline to make general mathematical calculations is a difficult and tedious task. The lack of signed arithmetic and presence of clamping in the pipeline necessitated quite some effort to overcome. In addition, fitting operations into a pipeline is a somewhat tricky task as it often isn't clear what should be calculated on which pass in order to maximize efficiency. Of course, when a task is tedious, one would like to relegate it to our dumb workhorse — the computer. Thus a graphics compiler would help programmers enormously.

A graphics compiler would be an ambitious project since there is a large variety of hardware which is currently changing very rapidly. However, given sufficient manpower, a compiler for a range of hardware could be created and maintained. This would give programmers a higher–level API to more easily implement their mathematical calculations using the graphics hardware.

Finally

It is clear from this thesis that graphics hardware can be used for more general calculations, even when it wasn't designed to do so. This leads one to believe that an intentional design process will yield the next generation graphics hardware: a SIMD–like machine capable of more generalized hardware calculations.

Bibliography

[1] Kurt Akeley. Realityengine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 109–116, 1993.

[2] James Arvo. Backward ray tracing. *Developments in Ray Tracing. ACM SIGGRAPH Course Notes*, 12:259–263, 1986.

[3] David C. Banks. Illuminations in diverse codimensions. *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 327–334, July, 1994.

[4] G. Barton. *Elements of Green's Functions and Propagation*. Clarendon Press, 1989.

[5] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292, August 1978.

[6] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546, 1976.

[7] David Blythe, Brad Grantham, Mark J. Kilgard, Tom McReynolds, and Scott R. Nelson. Advanced graphics programming techniques using opengl. In *SIGGRAPH 99 Course Notes*, August, 1999.

[8] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 263–272, August 1993.

[9] Eric Shencheng Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner. A progressive multi-pass method for global illumination. *Computer Graphics*, 25(4):164–174, 1991.

[10] Steven Collins. Adaptive splatting for specular to diffuse light transport. *proceedings of 5. Eurographics Workshop on Rendering*, pages 119–135, Darmstadt 1994.

[11] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, July 1977.

[12] Paul J. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering.* PhD thesis, U. of Pennsylvania, 1996. http://www.cis.upenn.edu/ diefenba/home.html.

[13] Paul J. Diefenbach and Norman I. Badler. Pipeline rendering: Interactive refractions, reflections, and shadows. *Displays*, 15(3):173–180, 1994. http://www.cis.upenn.edu/ diefenba/home.html.

[14] Paul J. Diefenbach and Norman I. Badler. Multi-pass pipeline rendering: Realism for dynamic environments. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 59–70. ACM SIGGRAPH, April 1997. ISBN 0-89791-884-3.

[15] I. Ernst, H. Rüsseler, H. Schulz, and O. Wittig. Gouraud bump mapping. In *Eurographics/SIGGRAPH Workshop on graphics hardware*, pages 47–54, 1998.

[16] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. *Fourth Eurographics Workshop on Rendering*, June:259–266, 1993.

[17] Eugene Hecht and Alfred Zajac. *Optics.* Addison Wesley Longman, Inc., 1979.

[18] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics*, 24(4):145–154, 1990.

[19] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *SIGGRAPH '84 Proceedings, pages 119–127*, July, 1984.

[20] Wolfgang Heidrich. Personal communication.

[21] Wolfgang Heidrich. *High–quality Shading and Lighting for Hardware–accelerated Rendering*. PhD thesis, University of Erlangen–Nurenberg, April, 1999.

[22] Wolfgang Heidrich, Jan Kautz, Philipp Slusallek, and Hans-Peter Seidel. Canned lightsources. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, 1998.

[23] Wolfgang Heidrich, Hendrick Lensch, Michael F. Cohen, and Hans-Peter Seidel. Light field techniques for reflections and refractions. In *Rendering Techniques '99 (Proceedings of the Eurographics Rendering Workshop)*, August 1999.

[24] Wolfgang Heidrich and Hans-Peter Seidel. Efficient rendering of anisotopic surfaces using computer graphics hardware. *Proceedins of the Image and Multi-dimentional Digital Signal Processing Workshop (IMDSP)*, 1998.

[25] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.

[26] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, August,1999.

[27] Wolfgang Heidrich, Rudiger Westermann, Hans-Peter Seidel, and Thomas Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, August1999.

[28] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. *Computer Graphics (SIGGRAPH '99 Proceedings)*, pages 277–286, August 1999.

[29] Silicon Graphics Inc. Pixel texture extension. Specification, http://www.opengl.org.

[30] Henrik Wann Jensen. Rendering caustics on non-lambertian surfaces. *Computer Graphics Forum*, 16(1):57–64, 1997.

[31] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real–time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics*, 24(4):327–335, August, 1990.

[32] Michael D. McCool and Wolfgang Heidrich. Texture shaders. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1999.

[33] Gavin Miller, Mark Halstead, and Michael Clifton. On–the–fly texture computation for real–time surface shading. *IEEE Computer Graphics & Applications*, 18(2):44–58, 1998.

[34] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinitereality: a real-time graphics system. In *Computer Graphics (SIGGRAPH '97 Proceedings), pages 293–302*, August,1997.

[35] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 303–306, August 1997.

[36] Bui-Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18:311–317, 1975.

[37] Marc Segal, Carl Korobkin, Rolf van Windenfelt, Jim Foran, and Paul Haeberli. Fast shadow and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July, 1992.

[38] Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 1.2.1). April, 1999.

[39] Cyril Soler and François X. Sillion. Fast calculation of soft shadow textures using convolution. *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 321–332, July 1998.

[40] Detlev Stalling, Malte Zöckler, and Hans-Christian Hege. Fast display of illuminated field lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, 1997.

[41] Jos Stam. Random caustics: Natural textures and wave theory revisited. In *ACM SIGGRAPH Visual Proceedings*, page 151, 1996.

[42] Jerry Tessendorf. Simulating ocean water. In *SIGGRAPH 99 Course Notes*, August, 1999.

[43] Keeneth E. Torrance, E. M. Sparrow, and R. C. Birkebak. Polarization, directional distribution, and off-specular phenomena in light reflected from roughened surfaces. *Journal of the Optical Society of America*, 56(7):916–925, 1966.

[44] Keeneth E. Torrance, E. M. Sparrow, and R. C. Birkebak. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, 1967.

[45] D. Voorhies and J. Foran. Reflection vector shading hardware. *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July, 1994.

[46] Mark Watt. Light-water interaction using backward beam tracing. *Computer Graphics*, 24(4):327–335, 1990.

[47] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

[48] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison Wesley Longman, Inc., third edition, 1999.