General calculations using graphics hardware, with application to interactive caustics

Chris Trendall and A. James Stewart
iMAGIS–GRAVIR/IMAG and University of Toronto

Abstract. Graphics hardware has been developed with image production in mind, but current hardware can be exploited for much more general computation. This paper shows that graphics hardware can perform general calculations, which accelerate the rendering process much earlier than at the latter image generation stages. An example is given of the real time calculation of refractive caustics.

1 Introduction

There is much more to graphics than meets the eye — the final image is usually the result of a long series of calculations. Many of these calculations are not directly related to images, but are of a general mathematical character.

Graphics hardware has been developed with image production in mind, but current hardware allows for much more general computation. This paper considers how far back along the long series of calculations the graphics hardware can contribute, and concludes that it can be used for much more than the image generation operations that are currently used.

If current hardware — designed for specific lighting effects rather than general calculations — is already this powerful, it is well worth considering what the next–generation hardware might be able to do if pushed in this direction. This paper considers the limitations of current hardware and discusses potential improvements that could greatly accelerate a more general set of calculations commonly used in graphics.

The paper's contributions are (a) to show how general calculations can be done with current hardware, to evaluate the shortcomings, and to suggest improvements for the future, and (b) to show one example of such a calculation, that of refractive caustics on a pool bottom due to light passing through a time–varying surface.

The paper shows that a time–varying heightfield can be interactively created, and that the refraction, scattering, absorption, and integration of light from this surface can be calculated to render the refractive caustics at the bottom of the pool in real–time.

The real–time computation of realistic caustics is an interesting problem in itself as computing specular lighting in dynamic environments can be computationally expensive. This is the first example of real–time refractive caustic generation.

2 Related 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 recent hardware–based calculations.

Heidrich has detailed methods to implement a wider variety of reflection models using multi–pass methods on existing hardware [10, 12, 14], and has discussed the more flexible parabolic parameterization of environment maps [10, 13] which allows multiple viewing directions. His general approach is similar to the approach in this paper: A set of carefully chosen mathematical operations can be combined in order to make a much broader range of calculations.

McCool and Heidrich suggest some basic operations which could be implemented in hardware to support a flexible per–vertex lighting model [10, 19]. Heidrich also shows how specifying a normal map permits the efficient calculation 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 [10, 11]. In this scheme, geometry information is coded into texture maps which treats the texture as a mathematical object rather than a visual one.

Miné and Neyret [20] synthesize procedural textures with graphics hardware using OpenGL. They consider a specific case, and map the Perlin noise function into OpenGL.

Haeberli and Segal [6] show how texture mapping can be used to implement a number of effects, from anti–aliasing to volume rendering. Used in these manners, texture mapping can be thought of as a transfer function between geometry and image space.

In an application which is suggestive of the possibility of using graphics hardware to do more general mathematical calculations, Heidrich [15] shows how the line integral convolutions of a vector field introduced by Cabral *et al.* [4] and further developed by Stalling and Hege [22] can be fully implemented in current hardware. Although a line integral is a general mathematical operation, this 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 the present paper explores — more generalized mathematical calculations.

Recently there has been substantial interest in mapping general *shading* operations to graphics hardware. Shading language compilers have been proposed [7, 21] which treat graphics hardware as a SIMD machine and OpenGL as an assembly language.

Several researchers have used graphics hardware for mathematical calculations that are unconnected with image synthesis. Hoff *et al.* [16] leverage z buffering capabilities to calculate Voronoi diagrams, Lengyel *et al.* [18] perform real–time robot motion planning using rasterizing hardware, and Bohn [3] interprets a rectangle of pixels as a four dimensional vector function to do computation on a Kohonen feature map.

2.2 Refractive Caustics

Stam [23] 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 which are non–interactive and in which the viewer is unable to correlate the refracting surface with the refracted caustic.

Arvo [1] used backward ray tracing to calculate the position of photons emitted from a light source incident on an object in a scene. Interestingly, he treats texture maps as data structures by using them to accumulate illumination.

Heckbert and Hanrahan [9] leveraged the spatial coherence of polygonal objects by reflecting and refracting beams from the visible surfaces in the view frustum, starting at the eye point. The reverse approach was taken up by Watt [26], who calculated the caustics on the bottom of a swimming pool using backward beam tracing.

Both Watt's approach [26] 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 composed of n polygons. This is a result of their discretization. In the present paper, 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.

All the work discussed in Section 2.2 is neither interactive, nor achieved using hardware–based rendering.

3 Mathematical Capabilities of Graphics Hardware

Modern raster graphics implementations typically have a number of buffers with a depth of 32 bits per pixel or more. In the most general setting, each pixel can be considered to be a data element upon which the graphics hardware operates. This allows a single graphics language instruction to operate on multiple data as in a SIMD machine.

Since the bits associated with each pixel can be allocated to one to four components, a raster image can be interpreted as a scalar or vector valued function defined on a discrete rectangular domain in the xy plane. The luminance value of a pixel can 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 can 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.

One of the drawbacks of current hardware is that only limited precision can be expected due to the limited depth of the buffers. The other major drawback is the lack of signed arithmetic, which leads to much effort in choosing scaling and biasing coefficients to simulate signed arithmetic. This is particularly troublesome as the rendering pipeline clamps the element values to $[0, 1]$ in a number of places.

3.1 Imaging Pipeline

For concreteness, the functionality of OpenGL 1.2 and some of its experimental extensions will be considered. The functions mentioned are all in the rasterization and per–fragment areas of the pipeline, and are performed by moving rectangles of pixels through the imaging pipeline. The details of these functions can be found in [17, 27].

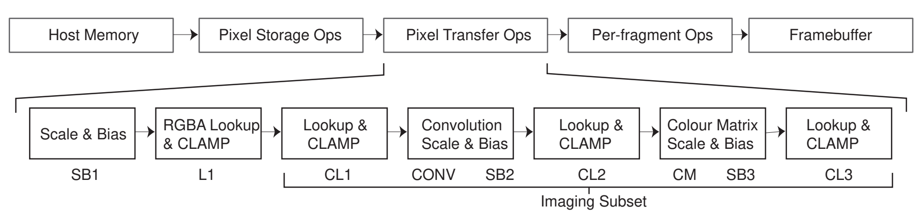

Fig. 1. OpenGl 1.2 imaging pipeline (top) and pixel transfer section (bottom)

The *Scaling and biasing* operation provides a linear transformation, *colour tables* provide arbitrary functions of one variable, a *colour matrix* provides affine transformations, and a *discrete convolution* gives a mechanism for approximating some integrals.

These operations are found in the imaging subset of OpenGL 1.2. [27]

Pixel texturing [17] is part of the per–fragment operations, and performs a mapping between a domain and range of up to four dimensions each. This provides the ability to implement a large class of functions. While this extension is not part of OpenGL 1.2, it exists on several SGI renderers, and equivalent functionality is becoming available on consumer level graphics hardware.

Blending is also part of the per–fragment operations, and provides a mechanism for calculating the product, sum, or difference of functions defined on a rectangular two–dimensional domain.

3.2 Mathematical Operations

General mathematical operations can be built on the hardware capabilities described in the previous section. The mathematical operations detailed below are all used in the refractive caustic demonstration.

Of the possible operations: *convolutions* with limited kernels can be computed directly; *derivatives* of height fields can be computed either by convolution [27] or by blending in the accumulation buffer [2]; *height field normals* can be computed using pixel texturing to normalize the height field derivatives; and *dot products* of a vector field with up to four constant vectors can be computed by the colour matrix. More extensive support for dot products can be found in recent hardware, such as in the NVIDIA register combiner architecture [5].

In order to implement *signed arithmetic* and avoid clamping, functions must be scaled and biased into the range $[0, 1]$. Since clamping always occurs after CL3, BLEND, and prior to enabled colour lookups CLx (see Fig. 1), all values must be properly scaled and biased by the end (respectively beginning) of these operations.

Let $\widetilde{f}$ represent a function f that has been scaled and biased: $\widetilde{f} = \alpha\, f + \beta$. Such a function is said to have been *adjusted*.

Although proper scaling and biasing requires one to know the approximate bounds of computation beforehand, this is a situation which always exists in computing as all datatypes are finite. It is a somewhat less restrictive issue when the datatype can accommodate a wider range of values but overflow and underflow are always potentially problematic.

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.) In some cases, the procedure differs slightly depending on the location of the data — framebuffer or main memory.

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:

$$\widetilde{f + c} \;=\; \alpha\,(f + c) + \beta \;=\; \alpha\, f + \beta + \alpha\, c \;=\; \widetilde{f} + \alpha\, c.$$

To perform this in hardware, it is sufficient to set the bias to $\alpha\, c$ at some point in the pipeline and to send $\widetilde{f}$ through the pipeline.

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, we must scale g by α and use additive blending. 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:

$$\widetilde{f+g} \;=\; \alpha(f+g)+\beta \;=\; \alpha f+\beta+\alpha g+\beta-\beta \;=\; \widetilde{f}+\widetilde{g}-\beta.$$

Thus β must be subtracted from the sum either before or after additive blending, or must be added to the difference after subtractive blending.

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:

$$\widetilde{cf} \;=\; \alpha\,(c\,f)+\beta \;=\; \alpha\,c\,f+\beta\,c+\beta-\beta\,c \;=\; c\,\widetilde{f}+(1-c)\,\beta.$$

This can be done by blending $\widetilde{f}$ with a pixel rectangle of constant value β using blend factors c and $1-c$, or by scaling and biasing $\widetilde{f}$ by c and $(1-c)\beta$.

This technique also works for vector functions. 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 scalar function g can be achieved in a similar manner. Again we have

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

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.

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

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

Multiplication of two adjusted functions in this manner is quite expensive, requiring three passes through the pipeline.

4 Refractive Caustics

This section demonstrates a general calculation using the graphics hardware. Refractive caustics will be computed on the bottom plane of a square pond with a time–varying surface height field, as shown in Figure 1 (see Appendix).

Consider light incident on a particular point on the surface, arriving from direction $\hat{\mathbf{L}}_\mathbf{i}$. The direction $\widehat{\mathbf{L}}_\mathbf{t}$ of transmitted light is given by Snell's Law [24]:

$$\widehat{\mathbf{L}}_\mathbf{t} = \frac{n_i}{n_t}\,\hat{\mathbf{L}}_\mathbf{i} + \Gamma\,\widehat{\mathbf{N}}, \text{ where } \Gamma = \frac{n_i}{n_t}\cos\theta_i - \cos\theta_t, \tag{1}$$

where n_i and n_t are the absolute indices of refraction in air and water, respectively, $\widehat{\mathbf{N}}$ is the surface normal, and θ_i and θ_t are the angles of incidence and transmission with respect to $\widehat{\mathbf{N}}$. Each of $\widehat{\mathbf{N}}$, $\hat{\mathbf{L}}_\mathbf{i}$, and $\widehat{\mathbf{L}}_\mathbf{t}$ is normalized.

If $L_i\,\delta(\theta - \theta_i)$ is the radiance incident on some infinitesimal surface, 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 [8]:

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

The effect of multiple scattering in the forward direction is empirically modelled by a Gaussian distribution in angle about $\widehat{\mathbf{L}}_\mathbf{t}$. This Gaussian models the effect of scattering throughout the volume; any scattering that ends up sending light in a forward direction (including photons that backscatter and then reverse direction to go forward, even multiple times) is taken into account in this model:

$$L(\,\widehat{\mathbf{P}}\,) \sim \exp(-a(1 - (\widehat{\mathbf{L}}_\mathbf{t}\cdot\widehat{\mathbf{P}})^2)).$$

Above, $L(\widehat{\mathbf{P}})$ is the radiance in (normalized) direction $\widehat{\mathbf{P}}$ and a is a coefficient proportional to the density of particulate matter, which is dependent on the rate of absorption and multiple backscattering.

Attenuation due to absorption is modelled as

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

where $L(\,d\,)$ is the radiance at distance d travelled by the light and ε is the *extinction coefficient*, which is dependent on the absorption and total backscatter in the volume. This type of extinction occurs when the absorption probability is constant with position [24].

Let L_s be the radiance of the point source, which is assumed to be very distant from the surface. Then the incident direction $\hat{\mathbf{L}}_\mathbf{i}$ is constant over the surface. Assuming that the surface is nowhere self–shadowing, the irradiance on the bottom plane can be calculated in this model by integrating over the surface of the air–water boundary. If we assume that the bottom 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 surface 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

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

Evaluation of this integral at discrete points on the bottom plane yields an image of the caustic induced by the height field.

4.1 Discretization and Approximation

The problem to be solved is to evaluate equation (2), 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:

- The surface is represented by a height field h which is a discrete sampling of the continuous surface. It is represented as a texture map $\tilde{h}$ of values in the range $[0, 1]$ with 0.5 representing the zero–height of the surface. The normal $\widehat{N}_{i,j}$ at $h_{i,j}$ is the normal to the plane passing through $h_{i,j}$, $h_{i+1,j}$, and $h_{i,j+1}$, and is computed with finite differences, then normalized with a lookup table.
- The Fresnel transmission coefficient $T(n_i, n_t, \cos\theta_i, \cos\theta_t)$ is discretized as a function of $\cos\theta_i$ and evaluated with a lookup table. This is possible since $\cos\theta_t$ is a function of $\cos\theta_i$, and n_i, n_t are constant.
- In the same manner, the Γ term in equation 1 is discretized as a function of $\cos\theta_i$ and evaluated with a lookup table.

Since the imaging pipeline has the ability to perform convolutions, the integral of equation (2) 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 expressed as $\int f(x, y)\; g(x - x_0, y - y_0)\; dx\, dy$ so that g can be used as a convolution kernel applied to f. 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 equation (2), we have the following components:

- Since $\theta_i = -\hat{\mathbf{L}}_\mathbf{i} \cdot \widehat{\mathbf{N}}$ and $\widehat{\mathbf{N}}$ is a function of $\frac{\partial h(x,y)}{\partial x}$ and $\frac{\partial h(x,y)}{\partial y}$, $T = T(x, y)$ is trivially separable. Similarly, $\theta_i = \theta_i(x, y)$, and $\cos\theta_i$ is trivially separable.
- We make the approximation $d \gg h(x, y)$ so that $-d - h(x, y) \approx -d$. Then $\exp(-\varepsilon\|\vec{\mathbf{P}}\|)$ and $\widehat{\mathbf{P}}_z$ become functions of $(x_0 - x, y_0 - y)$ and are both trivially separable.

Finally, an approximation is needed for the Gaussian, $\exp(-a(1 - (\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^2))$. We use an exponentiated cosine, $(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^n$. Keeping the terms to first order in $\widehat{\mathbf{L}}_{\mathbf{t}x}$ and $\widehat{\mathbf{L}}_{\mathbf{t}y}$ in the multinomial expansion for $(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^n$ gives

$$(\widehat{\mathbf{L}}_\mathbf{t} \cdot \widehat{\mathbf{P}})^n \approx (\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z)^{n-1}(\widehat{\mathbf{L}}_{\mathbf{t}z}\widehat{\mathbf{P}}_z + n(\widehat{\mathbf{L}}_{\mathbf{t}x}\widehat{\mathbf{P}}_x) + n(\widehat{\mathbf{L}}_{\mathbf{t}y}\widehat{\mathbf{P}}_y)),$$

which is most accurate for $\widehat{\mathbf{L}}_{\mathbf{t}z} \gg \widehat{\mathbf{L}}_{\mathbf{t}x}, \widehat{\mathbf{L}}_{\mathbf{t}y}$. 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 the result is the sum of three separable functions.

Combining these results and discretizing makes L a sum of three convolutions:

$$L_{1j} \quad = \quad L_s \sum\sum \left[T(\cos\theta_i)\cos\theta_t\, (\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1}\, \widehat{\mathbf{L}}_{\mathbf{t}j}\right] \left[(\widehat{\mathbf{P}}_z)^n\, \widehat{\mathbf{P}}_j\, \exp(-\varepsilon\|\vec{P}\|)\right],$$

where $j = x, y, z$. The first term in each is the kernel, and the double sum is over the size of the kernel.

4.2 Hardware Algorithm

The caustic computation algorithm performs the following steps:

Init: Calculate lookup tables, convolution kernels
FOR EACH HEIGHT FIELD IN THE SEQUENCE OF TIME–VARYING HEIGHT FIELDS
  Pass 1: Calculate $\widehat{\mathbf{N}}_{ij}$, vector field of normals to height field
  Pass 2: Calculate $\Gamma\,\widehat{\mathbf{N}}_{ij}$, a term need to determine $\widehat{\mathbf{L}}_{\mathbf{t}}$
  Pass 3: Calculate $L_s \cos\theta_i\, T(\cos\theta_i)$
  Pass 4: Calculate $\widehat{\mathbf{L}}_{\mathbf{t}}\,(\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1}$
  Pass 5: Calculate function to be convolved: $(L_s \cos\theta_i\, T(\cos\theta_i)) \cdot (\widehat{\mathbf{L}}_{\mathbf{t}}\,(\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1})$
  Pass 6: Convolve function using precomputed kernel
REPEAT

In *Pass 1* the heightfield normals are calculated by convolving to get the x and y discrete partial derivatives, and then using pixel texturing to look up the associated normal.

In *Pass 2*, the adjusted function $\widetilde{\Gamma\,\widehat{\mathbf{N}}}$ is calculated by multiplying the adjusted function $\widetilde{\widehat{\mathbf{N}}}$, by an unadjusted scalar function, Γ as described in Section 3.2. In order to achieve this, a pixel rectangle containing the bias value of the adjusted function must first be copied to the framebuffer before blending.

In *Pass 3*, $L_s \cos\theta_i\, T(\cos\theta_i)$ is calculated by multiplying an unadjusted function, $\cos\theta_i T(\cos\theta_i)$, by a function, L_s[1]. Since each of the terms is in the range $[0, 1]$, neither scaling nor biasing is necessary, and the result is an unadjusted function.

In *Pass 4*, $\widehat{\mathbf{L}}_{\mathbf{t}}\,(\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1}$ is calculated by multiplying an adjusted function, $\widetilde{\widehat{\mathbf{L}}}_{\mathbf{t}}$ (see Eq. 1), by an unadjusted function, $(\widehat{\mathbf{L}}_{\mathbf{t}z})^{n-1}$.

In *Pass 5*, the function F to be convolved is calculated. F is the product of the results of Passes 3 and 4 — an adjusted function and an unadjusted function.

In *Pass 6*, the three final convolutions are performed and the results summed in order to give the irradiance, E, on the bottom plane.

4.3 Heightfield Generation

This section describes the real time generation of a height field which represents the surface of the pond. The user can click to create an impulse at a point on the surface (as though dropping a pebble). Over time, the wave propagates, attenuates, and reflects off of the pond walls. Multiple waves are possible with multiple impulses.

Wave height h is a function of time and distance from the original impulse:

$$h(\,t, d\,) = \exp(-\beta(f(t) - d)) \exp(-\alpha t) \sin(Kd - \omega t)$$

where $f(t)$ is the distance to the wavefront from the original impulse, α and β are damping constants, K is the wave number and ω the angular frequency. The β term attenuates the wave with distance behind the wavefront, while the α term attenuates it with time. This is a model of wave propagation in a fluid more viscous than water, but it may be replaced by any function of t and d without affecting the hardware computation described below.

[1] L_s can vary with surface position as in Fig. 2 (see Appendix).

We precompute and store two functions in textures: $h(t,d)$ and $T(u,v)$. The T texture stores distances from the impulse point as

$$T(u,v) = \sqrt{(u-c_u)^2 + (v-c_v)^2}$$

for (c_u, c_v) the center point of the texture. Then the contribution to the heightfield at position (x,y) due to an impulse at (i_x, i_y) is

$$h(t, T(x - i_x + c_u, y - i_y + c_v))$$

This contribution is easily calculated in hardware by first loading the colour map with one row of h: $h(t,\cdot)$. Then the appropriate rectangle of T is copied to the framebuffer. As the distances of T pass through the colour map, they are replaced with the corresponding heights. The functions h and T must of course be scaled and biased to the range $[0,1]$.

Multiple impulses are treated by accumulating the results of the individual impulses. Reflections off of the pond walls are simulated by placing, for each impulse, four additional impulses outside of the pond: Each additional impulse is placed at the reflection of the original impulse through one of the four pond walls. (The dimensions of T are four times those of the height field.)

5 Implementation Results

The hardware caustic computation algorithm was implemented in OpenGL on an SGI Indigo 2 with a High Impact renderer and on an SGI Infinite Reality with an Onyx 2 renderer. A separate, completely software implementation was also made to validate the hardware results and to compare frame rates. The caustic algorithms were tested on sequences of height fields generated by the algorithm of Section 4.3. The resulting animation can be found at [25].

The Infinite Reality does not support pixel texturing and, despite the documentation to the contrary, we were unable to enable pixel texturing on the Indigo 2. The pixel texturing was thus simulated in software.

5.1 Frame Rates

Table 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, and the software implementation.

The frame rates of Table 1 are those required to generate the raster images of caustics from raster images of height fields. Running times for height field generation on an Infinite Reality Onyx 2 are 178 frames per second for a 128×128 heightfield with one impulse (plus four reflected impulses), and 64 frames per second for a 256×256 heightfield with three impulses (plus 12 reflected impulses).

From Table 1, it is clear that interactive rates of between 9 and 15 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.

Table 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. HW/No PT: Hardware implementation , pixel texturing disabled. HW/SWPT: Hardware implementation with software pixel texturing. SW: Software implementation. Sizes are in pixels.

	Infinite Reality			Indigo2		
Size	HW/No PT	HW/SWPT	SW	HW/No PT	HW/SWPT	SW
128×128	37.9	24.8	2.56	11.9	8.6	1.0
256×256	14.9	9.4	0.6	4.2	2.6	0.2

5.2 Correctness

The caustics resulting from one of the more complex height fields were calculated in both software and hardware in order to compare the numerical values of the irradiance. The software implementation made the same approximations as the hardware implementation, but the calculations in software were done in floating point so that the loss of precision due to the limited depth and discretization imposed by the hardware could be measured. For the height field seen in Fig. 1 (see Appendix), the relative RMS error between the two caustics was 26%. The height field which was used is clearly not the the height field in the sequence for which the error is minimal, and was chosen to estimate the worst case bound on the precision error.

6 Discussion

This paper has shown that graphics hardware can be used to perform complex general calculations, and has demonstrated an example of the calculation of interactive caustics. Although the calculation is clearly not as precise as a floating point implementation, there are many situations where current hardware can be used to make such calculations. The likelihood of deeper hardware buffers in the future make this an even more important area of research, as the number of problems that can be approached with these techniques will grow.

The calculation of caustics is particularly suitable for an implementation which lacks precision since visual perception of tones and shades is much less precise than for position, for example. There are many problems in which precision is unnecessary such as initial approximations to mathematical equations, situations where perceptual acuity is limited due to distance, motion, or lighting conditions, and the description of natural phenomena that appear to have an element of randomness. Generally speaking, when speed of calculation is more important than precision, mathematics on current graphics hardware can be very useful.

An increase in numerical precision would, of course, extend the set of problems amenable to this treatment. The implementation of floating point operations in graphics hardware, for example, would greatly enhance the applicability of graphics math. With or without floating point calculations, deeper buffers would increase numerical precision but at the cost of bandwidth. In addition, in lieu of full floating point, a floating point scale and bias prior to the initial quantization to framebuffer resolution would allow for increased dynamic range.

The approximations necessary to implement caustic generation in hardware provide a good clue as to the features which would improve such computations. The most significant approximations involved integration as current hardware is equipped with only a relatively small convolution kernel for implementing this operation. Certainly larger kernels would increase the applicability of this method, but larger kernels are likely to reduce performance, and the size of the kernel is only part of the difficulty: in order to implement integration as convolution, the integrand must be separable. To ease this restriction, operations on the kernel during the convolution would be useful, especially after the multiplication operation and before the summation.

Any reduction of clamping in the pipeline would also be beneficial. A 'wrap-around mode' for table lookups would allow for modular arithmetic, for example, and currently a lookup causes all the colour components to be clamped, even if they aren't used for the lookup, which could be prevented. Lookups are also currently implemented by quantizing and taking the nearest value in the table. This is quite appropriate for linear or near–linear functions with small dynamic range, but highly non–linear functions lose more precision in this algorithm. Interpolation would be a possible remedy to extend the applicability of lookup tables, but one must be careful not to sacrifice too much performance.

Although there is currently some capability in OpenGL for conditional execution such as stencil buffering and min/max functions, extending the scope and functionality of these operations would provide much greater flexibility. This direction has been taken in the register combiner architecture of NVIDIA, where different general combiner computations are allowed depending on one of the inputs to a combiner stage [5].

Programming the graphics hardware to do general mathematics can be quite tedious, which suggests that a computational approach might be suitable. Given a set of operations, software to aid fitting the operations into the pipeline would be quite useful. In addition, the tedium involved in correctly restricting values to $[0, 1]$ could be alleviated by an expression compiler which, given the types and ranges of the components of an expression, automatically computes the appropriate scale and bias. On a larger scale, one could imagine a 'graphics compiler' to analyse both scale and bias and pipeline fitting, and generate the necessary OpenGL code for the hardware in a similar manner to the shading language compilers recently proposed [7, 21].

Given the applications of graphics hardware to problems such as artificial intelligence [3], robot path planning [18], computational geometry [16], and now the creation and rendering of caustics, it is clear that graphics hardware can be used for much more general mathematical purposes than for which it was first intended: graphics hardware is not just for images any more.

6.1 Acknowledgements

We would like to thank all the people at iMAGIS–GRAVIR/IMAG for their kindness during our stay there, especially David Bourguignon for the donation of his machine. Special thanks also to David Torre for his help preparing images.

References

1. James Arvo. Backward ray tracing. *Developments in Ray Tracing. ACM SIGGRAPH Course Notes*, 12:259–263, 1986.
2. 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.
3. Christian-A. Bohn. Kohonen feature mapping through graphics hardware. In *3rd Int. Conf. on Computational Intelligence and Neurosciences*, 1998.

4. Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 263–272, August 1993.
5. NVIDIA Corporation. Nvidia OpenGL Extension Specifications. http://www.nvidia.com, April, 2000.
6. Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. *Fourth Eurographics Workshop on Rendering*, June:259–266, 1993.
7. Pat Hanrahan, Bill Mark, Kekoa Proudfoot, Svetoslav Tzvetkov, David Ebert, Wolfgang Heidrich, and Philipp Slusallek. Stanford Real–Time Programmable Shading Project. http://graphics.stanford.EDU/projects/shading/, 2000.
8. Eugene Hecht and Alfred Zajac. *Optics*. Addison Wesley Longman, Inc., 1979.
9. Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *SIGGRAPH '84 Proceedings*, pages 119–127, July, 1984.
10. Wolfgang Heidrich. *High–quality Shading and Lighting for Hardware–accelerated Rendering*. PhD thesis, University of Erlangen–Nurenberg, April, 1999.
11. 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.
12. Wolfgang Heidrich and Hans-Peter Seidel. Efficient rendering of anisotopic surfaces using computer graphics hardware. *Proceedings of the Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP)*, 1998.
13. Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.
14. Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, August,1999.
15. 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*, August, 1999.
16. 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.
17. Silicon Graphics Inc. Pixel texture extension. Specification, http://www.opengl.org, 1999.
18. 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.
19. Michael D. McCool and Wolfgang Heidrich. Texture shaders. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1999.
20. A Miné and F Neyret. Perlin textures in real time using OpenGL. Technical Report RR-3713, iMAGIS–GRAVIR/IMAG/INRIA, 1999.
21. Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi–pass programmable shading. In *Computer Graphics (SIGGRAPH 2000 Proceedings)*, July, 2000.
22. 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.
23. Jos Stam. Random caustics: Natural textures and wave theory revisited. In *ACM SIGGRAPH Visual Proceedings*, page 151, 1996.
24. Jerry Tessendorf. Simulating ocean water. In *SIGGRAPH 99 Course Notes*, August, 1999.
25. Chris Trendall and James Stewart. An example of hardware mathematics: Refractive caustics. http://www.dgp.utoronto.ca/˜trendall/egwr00/index.html, 2000.
26. Mark Watt. Light-water interaction using backward beam tracing. *Computer Graphics*, 24(4):327–335, 1990.
27. Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison Wesley Longman, Inc., third edition, 1999.

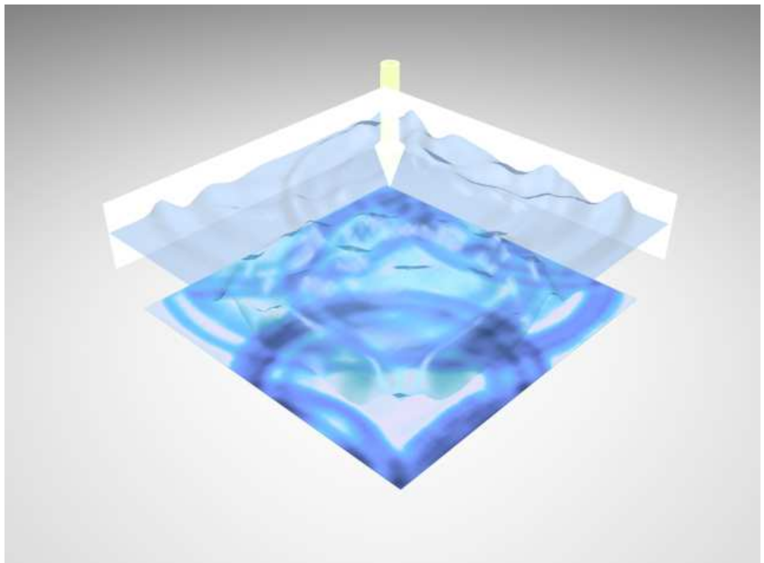

Fig. 2. The water surface/height field is shown in transparent blue, bounded by two white pool borders in a cutaway view. The angle of the incident light is shown by the yellow arrow. The resulting caustics appear on the floor plane below the surface.

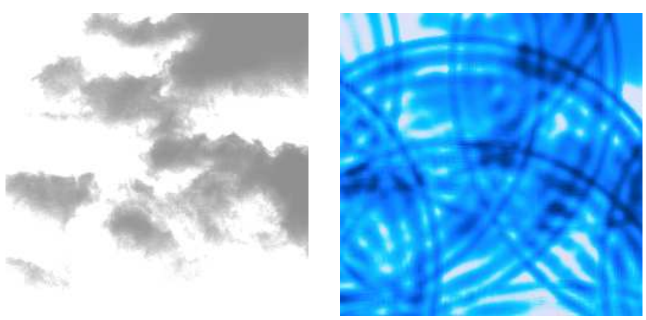

Fig. 3. The spatially varying surface illumination on the left gives rise to the caustics on the right.