Interactive Relighting of Dynamic Refractive Objects

Xin Sun* Kun Zhou† Eric Stollnitz ‡ Jiaoying Shi* Baining Guo†

*Zhejiang University †Microsoft Research Asia ‡Microsoft Research

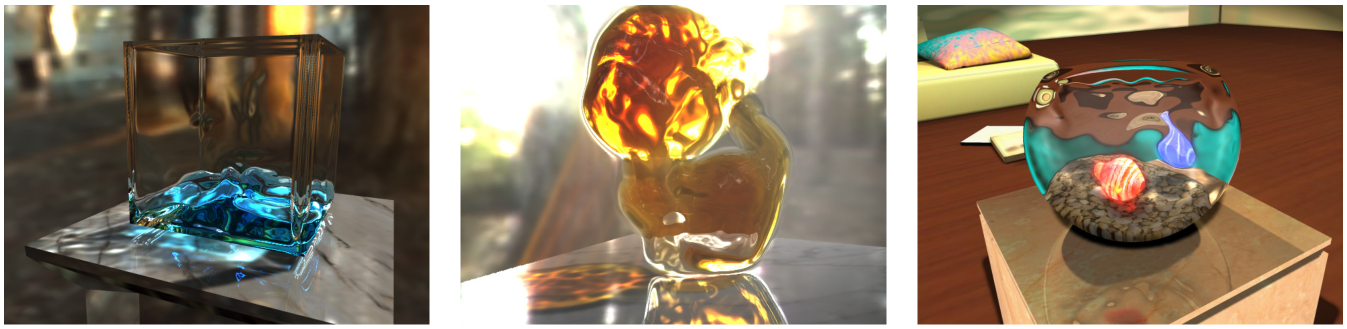

(a) (b) (c)

Figure 1: *Renderings produced by our technique at interactive rates.*

Abstract

We present a new technique for interactive relighting of dynamic refractive objects with complex material properties. We describe our technique in terms of a rendering pipeline in which each stage runs entirely on the GPU. The rendering pipeline converts surfaces to volumetric data, traces the curved paths of photons as they refract through the volume, and renders arbitrary views of the resulting radiance distribution. Our rendering pipeline is fast enough to permit interactive updates to lighting, materials, geometry, and viewing parameters without any precomputation. Applications of our technique include the visualization of caustics, absorption, and scattering while running physical simulations or while manipulating surfaces in real time.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: refractive objects, ray tracing, photon tracing, interactive relighting

1 Introduction

The refraction and scattering of light as it passes through different materials results in many beautiful and intriguing effects, ranging from mirages and sunsets to the caustic patterns produced by a crystal vase or water waves. Simulating these phenomena in computer graphics remains a challenge, particularly when the goal is to render at interactive rates in dynamic scenes where the lighting, materials, and geometry may all be changing. Users want to see the effects of refraction while animating objects, editing shapes, adjusting lights, and running physical simulations, but current techniques fall short.

The most interesting phenomena depend on indirect lighting, and therefore we need to solve a global illumination problem. In fact, the key challenge is to calculate a global illumination solution accurately enough to capture detailed caustics, yet quickly enough that we can change the lights, geometry, and materials in real time. Algorithms such as distribution ray tracing, Monte Carlo simulation, and photon mapping are widely used to produce accurate results, but not at interactive rates. Although the GPU has been used to significantly accelerate both photon mapping [Purcell et al. 2003] and wavefront-tracking techniques [Ihrke et al. 2007], the frame rates are still not sufficient for interactive relighting. There are several methods that generate approximate solutions in real time [Davis and Wyman 2007; Ernst et al. 2005; Krüger et al. 2006], but these approaches invariably neglect some of the important effects of refraction. In short, none of the existing techniques is well-suited to the interactive relighting of dynamic refractive objects with complex material properties.

In this paper, we describe a new technique for rendering scenes containing inhomogeneous refractive objects. Our approach is unique in its ability to render volumetric caustic, scattering, and absorption effects interactively, even as the scene changes. We tackle the problem by creating a new rendering pipeline, relying on the GPU for each of its stages in order to visualize these effects without any precomputation. The speed of our approach hinges on our use of a voxel-based representation of objects and illumination. We designed our rendering pipeline to start with on-the-fly voxelization, so that we can maintain a volumetric representation even when the input geometry is changing. Using this voxel-based representation allows us to fully exploit the GPU's parallelism in each of the subsequent pipeline stages.

For each frame, the pipeline voxelizes the input surfaces, traces photons to distribute radiance throughout the scene, and renders a view of the resulting radiance distribution. Since we permit continuous variation in the refractive index throughout the volume, photons and viewing rays follow curved paths governed by the ray equation of geometric optics. Because of our voxel-based representation, we can build an octree to further accelerate our algorithm; we use this octree to choose adaptive step sizes when propagating photons along their curved paths. We also leverage the GPU's ras-

† e-mail: kunzhou@acm.org, bainguo@microsoft.com
‡ e-mail: ericsto@microsoft.com

terization speed to update the stored radiance distribution in each step. As a result, our technique can render dynamic refractive objects such as those in Figure 1 at a rate of several frames per second.

The remainder of this paper is organized as follows: Section 2 briefly summarizes work related to ours. Section 3 provides an overview of our rendering pipeline, while Section 4 presents each of the pipeline stages in greater detail. Section 5 discusses some of the issues we faced in creating a GPU-based implementation of the pipeline. We present some results illustrating the use of our technique in Section 6, and conclude in Section 7.

2 Related Work

Many different techniques have been used to render refractive objects. Among these, perhaps the most common approach is to use Snell's law at boundaries between regions of constant refractive index. Nearly all renderers derived from conventional ray tracing [Whitted 1980] rely on this approach. Likewise, techniques that leverage the GPU for interactive display of refraction effects typically rely on Snell's law [Davis and Wyman 2007; Wyman 2005]. In contrast, Stam and Languénou [1996] describe how equations derived from geometric optics can be used to trace curved light rays through regions with non-constant index of refraction. Like Ihrke et al. [2007], we adopt this elegant approach for volumes containing inhomogeneous materials.

One of the important effects of refraction is the production of caustics—bright spots resulting from light rays being focused onto a surface or participating media. Photon maps, introduced by Jensen [1996] as a way to accelerate Monte Carlo global illumination algorithms, have proven to be excellent at rendering caustics. While photon maps were originally used to render surfaces, subsequent work on volume photon mapping [Jensen and Christensen 1998] describes how very similar techniques can also be used to simulate illumination effects in and around participating media, including anisotropic multiple scattering and volume caustics. Gutierrez et al. [2004; 2005] further extended volume photon mapping to the case of non-linear light rays passing through participating media of varying refractive index.

Much of the literature on photon mapping is devoted to off-line rendering, where the quality and accuracy of the result is valued over interactivity. However, Purcell et al. [2003] describe how photon mapping can be implemented on a GPU by using specialized data structures and algorithms that are amenable to parallel processing. Also relevant is the work by Weiskopf et al. [2004], which presents GPU-based techniques for accelerating non-linear ray tracing. Although these approaches achieve significant performance gains compared to the corresponding CPU-based algorithms, they fall short of delivering interactive frame rates, nor do they attempt to render volumetric effects.

Techniques that rely on image-space rendering of refractive effects have achieved impressive results, but many of these methods exclude caustics entirely [Davis and Wyman 2007; Oliveira and Brauwers 2007; Wyman 2005] or include surface caustics but not volume caustics [Szirmay-Kalos et al. 2005; Wyman and Davis 2006]. Truly accurate rendering of volume caustics requires a simulation of single and multiple scattering in participating media, as formulated by Kajiya and Von Herzen [1984]. Multiple scattering, which dominates in dense media like smoke and clouds, is beyond the scope of the present paper. Single scattering is dominant in nearly transparent materials like air and water. The shafts of light that are typical of single scattering have been approximated by blending layered materials [Dobashi et al. 2002], by warping volumes [Iwasaki et al. 2002; Ernst et al. 2005], and by compositing light beams into the image [Krüger et al. 2006]. These techniques

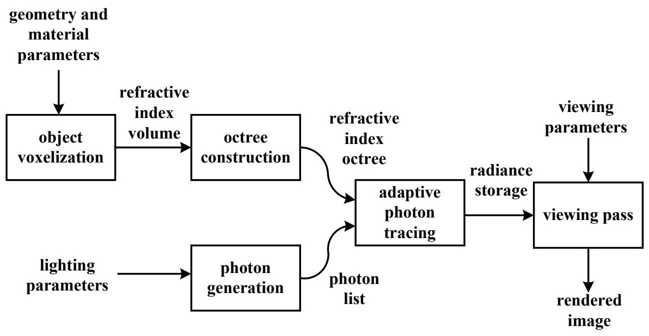


Figure 2: *Our rendering pipeline.*

tend to trade physical accuracy for the convenience of a fast GPU-based implementation. Sun et al. [2005] present a physically correct real-time technique for rendering single-scattering effects, but without refraction.

The algorithm presented by Krüger et al. [2006] renders many of the same effects as our approach, including single scattering and refraction with multiple interfaces. However, it doesn't account for absorption, and like all image-space methods, it fails to capture refractions outside the view frustum, making the caustics view-dependent. The "eikonal rendering" technique proposed recently by Ihrke et al. [2007] comes closest to rendering all the effects of refraction and participating media in real time. Although eikonal rendering can produce new views at interactive rates, any change to the lighting, materials, or geometry of the scene requires a recalculation of the radiance distribution that takes several seconds.

3 Overview of the Rendering Pipeline

We have developed a technique capable of rendering the effects of refraction, absorption, and anisotropic single scattering in dynamic scenes at interactive rates. Our rendering pipeline, depicted in Figure 2, consists of several stages, each of which is implemented on the GPU. The pipeline takes as its input a volumetric or surface description of the scene, a set of point or directional light sources, and the desired view; it produces a rendered image as its output. Note that during an interactive session, only some of the stages need to be executed if the application changes just the viewing parameters, just the lighting parameters, or just the geometry and materials.

Our technique relies on a volumetric representation of space in the form of a rectangular voxel grid; in particular, we require a voxel-based representation of the index of refraction n, the scattering coefficient σ, and the extinction coefficient κ (defined as the sum of σ and the absorption coefficient α). The first stage of our rendering pipeline harnesses the GPU to efficiently convert triangle-mesh surfaces into volumetric data. This voxelization stage can be bypassed in applications that obtain volumetric data directly from physical simulations or measurements.

The next stage of our pipeline analyzes the index of refraction data and produces an octree describing the regions of space in which the refractive index is nearly constant. This information, combined with a list of photons that we generate from the light sources, serves as input to the adaptive photon tracing stage of the pipeline. This stage advances each photon along its path through the scene, depositing radiance into all the voxels that the photon traverses. Finally, the viewing pass is responsible for rendering an image by tracing viewing rays from the camera into the scene and calculating the amount of radiance that reaches the camera along each ray.

Because we allow for media whose index of refraction varies continuously throughout space, both the photons and the viewing rays follow curved paths rather than the straight-line paths ordinarily used in photon mapping. A curved light path $\mathbf{x}(s)$ is related to the scalar field n of refractive index by the ray equation of geometric optics [Born and Wolf 1999]:

$$\frac{d}{ds}\left(n\,\frac{d\mathbf{x}}{ds}\right) = \nabla n\,. \tag{1}$$

By defining $\mathbf{v} = n\frac{d\mathbf{x}}{ds}$, we can rewrite Equation 1 as a system of first-order differential equations:

$$\frac{d\mathbf{x}}{ds} = \frac{\mathbf{v}}{n} \tag{2}$$

$$\frac{d\mathbf{v}}{ds} = \nabla n\,. \tag{3}$$

Our algorithm marches along piecewise-linear approximations to these curves using a forward-difference discretization of the continuous equations:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{\Delta s}{n}\,\mathbf{v}_i \tag{4}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \Delta s\,\nabla n\,. \tag{5}$$

Unlike previous work in this area, we change the step size Δs as we advance photons along these curves to adapt to the surrounding variation in refractive index. In a region of nearly constant refractive index, we can take a single large step, while in a region where the index varies greatly, we must take many small steps in order to accurately trace the light path. We use an octree data structure to determine how large a step we can take from any given position.

4 Rendering Pipeline Details

In the following sections, we describe in more detail each of the stages of the rendering pipeline depicted in Figure 2.

4.1 Object Voxelization

Several algorithms have been proposed for the conversion of surfaces into volume data. Among these techniques, some are unsuitable for our purposes because they only label voxels intersected by the surface and not those in the interior of the object [Eisemann and Décoret 2006; Zhang et al. 2007]. Our voxelization technique is based on the method presented by Crane et al. [2007, section 30.2.4], which is largely similar to that of Fang et al. [2000]. Their algorithm takes a watertight triangulated surface mesh as input and produces a volumetric texture as output, relying on the GPU's rasterization and clipping operations to assign a value of one to voxels whose centers are inside the mesh and zero to voxels whose centers are outside the mesh. This is accomplished by rendering the mesh into each slice of the volume texture in turn, with the near clipping plane set to the front of the slice, using a fragment shader that increments the voxel values within back-facing triangles and decrements the voxel values within front-facing triangles. We alter the original voxelization method slightly because our rendering pipeline, like eikonal rendering, requires smoothly varying values within the refractive index volume in order to avoid undesirable rendering artifacts.

We want the voxelization algorithm to assign fractional coverage values to those voxels through which the surface passes. We first super-sample the volume by voxelizing the mesh into a texture that is four times larger in each dimension than the output. We then need to down-sample the resulting texture, but the cost of reading $4 \times 4 \times 4 = 64$ texture samples for each of the output voxels can be prohibitive. Instead, we utilize a strategy that only requires down-sampling for output voxels near the surface. We voxelize the mesh

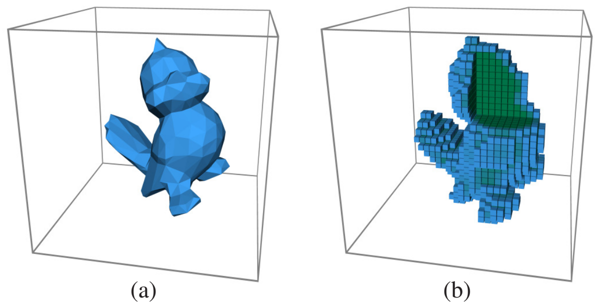


Figure 3: *(a) A triangle-mesh surface representation of an object. (b) The same object after conversion to volumetric data by our voxelization method (partially cut away to show the interior).*

again into a texture at the desired output resolution, then look at a $3 \times 3 \times 3$ neighborhood around each of the resulting voxels. If these 27 voxels all have the same value, no further work is required. If the voxel values differ, then we down-sample the corresponding region of the super-sampled texture to get the fractional value.

Finally, we convert the fractional coverage values into refractive index numbers and smooth the output texture using a Gaussian blur kernel. The Gaussian filter's window is $9 \times 9 \times 9$ voxels in the final voxel space, with a standard deviation that is one-sixth of the window size. Note that super-sampling effectively increases the accuracy of surface normals (as represented by the gradient of the refractive index), while blurring spreads the boundary region over a wider band of voxels. The result of our voxelization technique is illustrated for a typical mesh surface in Figure 3.

We determined our supersampling resolution and Gaussian blur kernel based on experimental renderings of a transparent sphere in front of a patterned background. We found that the quality of the refracted pattern improves as supersampling is increased, regardless of which blur kernel is used. Once we chose $4 \times 4 \times 4$ supersampling (the largest we deemed practical), we saw a small improvement as the blur kernel increased in size from 5 to 7 to 9, but no substantial change as the kernel size increased to 11. We therefore use $4 \times 4 \times 4$ supersampling and a $9 \times 9 \times 9$ kernel.

4.2 Octree Construction

The input to our octree construction algorithm is a tolerance value ε and a three-dimensional array containing the refractive index n for each voxel of a rectangular volume. The output is a representation of an octree within whose leaf nodes the refractive index is within ε of being constant. Because we are using the GPU rather than the CPU, we choose to represent the octree in a form that is appropriate for construction and access by multiple parallel processing units. Instead of the traditional sparse representation of an octree in which each internal node has pointers to eight child nodes, we output the octree as a dense three-dimensional array of numbers, where the value in each voxel indicates the hierarchy level of the leaf node covering that voxel. While this array may occupy more memory than a sparse tree would, it is much more easily constructed and accessed on a GPU.

The construction of our octree representation is similar to mipmap construction and to the process by which Ziegler et al. build histogram pyramids [2007]. Figure 4 illustrates the entire octree construction process, using a two-dimensional example for the sake of simplicity. We first construct a pyramid of 3D arrays that record the minimum and maximum refractive index present in each volumetric region, and then we use this pyramid along with the input tolerance ε to decide which level of the octree is sufficient to represent each of the original voxels.

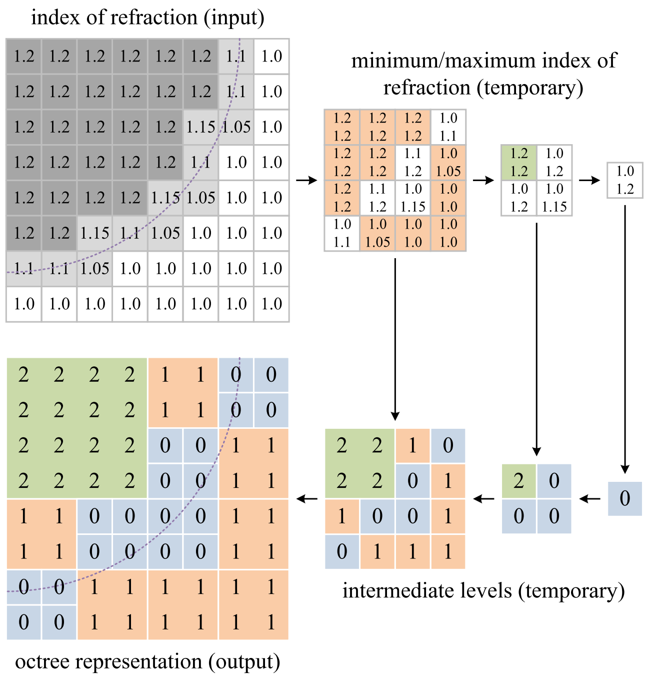

index of refraction (input)
minimum/maximum index of refraction (temporary)
intermediate levels (temporary)
octree representation (output)

Figure 4: *Octree construction algorithm, simplified to 2D. We first build a pyramid containing minimum and maximum refractive index values. We then analyze the pyramid from the coarsest level to the finest, writing the current level number into the octree representation whenever the range of refractive index values is within the input tolerance ε = 0.05.*

To be more precise, assume we are given a cube-shaped volume of refractive index entries n of size 2^K in each dimension. We start by building a pyramid of K three-dimensional arrays, much like a mipmap, except each element stores the minimum and the maximum index of refraction ($n_{\min}$ and $n_{\max}$) for the corresponding portion of the volume. Next, we construct another 3D array of size 2^K in each dimension, where each entry is an integer indicating the level of the octree sufficient to represent that portion of the volume. We use zero as the label for the finest level of the octree and K as the label for the coarsest (single-voxel) level of the octree. We determine the appropriate labels by iterating from the coarsest level to the finest level of the pyramid, comparing each range of refractive index to the input tolerance. While examining pyramid level k, as soon as we encounter an entry satisfying $n_{\max} - n_{\min} < \varepsilon$, we assign the corresponding volumetric region the label k. Once a voxel has been labeled with a level number, we do not label it again.

In principle, we should construct the octree in such a way that both the index of refraction n and the extinction coefficient κ are approximately constant across each leaf node, since our adaptive photon tracing algorithm is affected by both of these material properties. We can easily extend the octree construction algorithm to limit the variation in both properties within leaf nodes. In practice, however, we find that the visual impact of the extinction coefficient is not prominent enough to warrant the extra complexity.

4.3 Photon Generation

We generate the initial positions and velocities of photons using a method similar to shadow maps: we set up the graphics pipeline so that the camera is positioned at a light source and oriented toward the volume of interest, then render the cube faces that bound the volume. We draw into a texture, using a shader that records an alpha value of one along with the 3D position for each pixel representing a front-facing surface; empty pixels are left with alpha values of zero. Next, we transform this texture into a list of point primitives that represent photons, using either a geometry shader or a “scan” operation [Chatterjee et al. 1990] written in a general-purpose GPU programming language like CUDA [Nvidia Corporation 2007]. Each pixel with non-zero alpha produces a single photon, where the photon’s position is obtained from the pixel value, the photon’s direction is derived from the texture coordinates and light position, and the photon’s radiance is defined by the light’s emission characteristics.

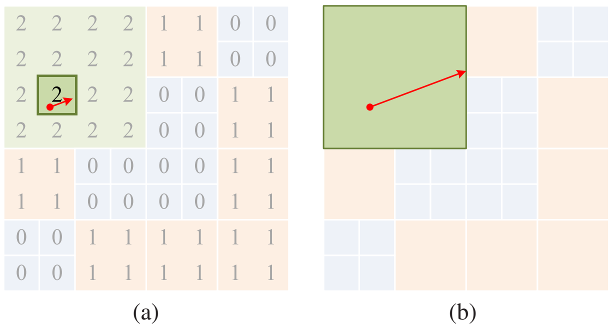


Figure 5: *Adaptive photon marching using an octree. (a) The photon’s position is used to determine the level of the surrounding leaf node of the octree. (b) The step size Δs_{octree} is calculated to advance the photon to the boundary of the octree node.*

When the scene consists of a solid transparent object, much of the volume surrounding the object consists of empty space, where the index of refraction is uniformly one. We can reduce the amount of time we will spend tracing photons through this empty space by generating the photons as close as possible to the interesting parts of the volume. To accomplish this, we can render a proxy geometry into the shadow map, rather than the bounding cube of the volume. Any bounding surface will do; for a complex object, we typically use a mesh that has been inflated to encompass the entire object.

4.4 Adaptive Photon Tracing

The goal of the adaptive photon tracing stage is to advance each photon along its curved path, while simultaneously depositing radiance into a volumetric texture for our later use. The input to this portion of our technique consists of our octree representation of the refractive index values, a 3D array of RGB extinction coefficients, and a list of photons. Each of the photons is equipped with an initial position $\mathbf{x}_0$, direction $\mathbf{v}_0$, and RGB radiance value $\tilde{L}_0$. The output is a 3D array of RGB radiance distributions describing the illumination that arrives at each voxel.

Each iteration of adaptive photon tracing marches the photons one step forward according to Equations 4 and 5. For each photon, we use the octree to determine the largest step size Δs_{octree} that keeps the photon within a region of approximately constant refractive index, as shown in Figure 5. We then choose Δs to be the larger of Δs_{octree} and a user-supplied minimum step size $\Delta s_{\min}$. The minimum step size is typically the width of one or two voxels, and can be adjusted by the user to trade off accuracy for performance. Our use of the octree to determine step sizes is reminiscent of the precomputed shells used by Moon et al. [2007] to determine step sizes when rendering multiple scattering effects.

In each step, a photon loses a fraction of its radiance to absorption and out-scattering. The rate of exponential attenuation is determined by the local extinction coefficient, which we assume to be approximately constant within the current octree node:

$$\tilde{L}_{i+1} = \tilde{L}_i \, e^{-\kappa(\mathbf{x}_i)\,||\mathbf{x}_{i+1}-\mathbf{x}_i||} \, . \tag{6}$$

We have described how to advance a photon by a single step, updating its position and direction according to Equations 4 and 5 and its radiance according to Equation 6. Now we turn to the accumulation of radiance distributions within the volume. In each voxel, we approximate the incoming radiance distribution by storing only a weighted average of the direction of arriving photons and a single radiance value for the red, green, and blue wavelengths. This is clearly a very coarse approximation of the true distribution of radiance, but it is sufficient to reproduce the effects we aim to render.

As a photon travels from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$, it should contribute radiance to each of the voxels through which it passes. Therefore, every time we advance a photon by a single step, we generate two vertices into a vertex buffer to record the photon's old and new position, direction, and radiance values. Once all the photons have been marched forward one step, we treat the vertex buffer as a list of line segments to be rasterized into the output array of radiance distributions. We rely on the graphics pipeline to interpolate the photon's position, direction, and radiance values between the two endpoints of each line segment. We use a pixel shader that adds the photon's radiance to the distribution stored in each voxel, and weight the photon's direction of travel by the sum of its red, green, and blue radiance values before adding it to the direction stored in each voxel.

After we have advanced all the photons and stored their contributions to the radiance distributions, we eliminate each photon that has exited the volume or whose radiance has fallen below a low threshold value, and then we repeat the entire process. We continue these iterations until the number of active photons is only a small fraction (1/1000, say) of the original number of photons. As a final step, we smooth the volume of radiance distributions using a $3 \times 3 \times 3$ approximation of a Gaussian kernel to reduce noise.

There are several key differences between our adaptive photon tracing and the approach of previous volume photon mapping algorithms. First and foremost, we use an adaptive step size when marching each photon through the volume; as mentioned earlier, our octree structure allows us to compute the longest step that remains within a region of nearly constant refractive index. Second, we use a voxel grid to store the radiance contributed by each photon to the illumination of the volume. We update the radiance distribution within every voxel that a photon passes through, rather than recording radiance values only when a scattering event occurs. Previous volume photon mapping algorithms rely on a kd-tree to store a sparse sampling of the radiance distribution, requiring a costly nearest-neighbor search when the viewing pass needs to reconstruct the radiance at other locations.

4.5 Viewing Pass

Once the photon tracing pass is complete, we can render images from arbitrary viewpoints. In the viewing pass, we trace backwards along the paths that light rays take to reach the camera, summing up the radiance contributed by each of the voxels that we traverse, after accounting for the scattering phase function and attenuation due to absorption and out-scattering.

We start by initializing the origin and direction of a single ray for each pixel of the output image. If the ray intersects the original triangle mesh (or a proxy geometry surrounding any refractive objects), we will march along the ray step-by-step until it exits the volume. We use Equations 4 and 5 to determine the paths taken by viewing rays, just as we did for photons. However, when tracing viewing rays, we use a fixed step size equivalent to the width of one voxel rather than an adaptive step size. There are several reasons to use a fixed step size: we don't want to miss the contribution of any of the voxels along the light ray's path; we don't want to introduce image artifacts caused by different step sizes among adjacent pixels; and from a practical standpoint, the viewing pass is so much faster than the adaptive photon tracing stage that it needs no acceleration.

At each step along a viewing ray, we look up the radiance value and direction stored in the corresponding voxel, then evaluate the scattering phase function to determine how much of the radiance is scattered from the incident direction toward the camera. We multiply the result by the local scattering coefficient $\sigma(\mathbf{x})$ and the total attenuation along the ray due to absorption and out-scattering. This product gives the radiance contribution of a single voxel, which we then add to the total radiance of the current viewing ray. Once the ray exits the volume, we need only incorporate any background radiance to complete the calculation of an output pixel color.

5 Implementation

We turn now to some issues that complicate our implementation. While it is important to point out these complexities, they are not essential to understanding the rendering pipeline.

5.1 Radiance Storage

Our GPU-based implementation of adaptive photon tracing divides the work into the photon marching pass, which calculates the new position, direction, and radiance of each photon after one step, and the photon storage pass, which accumulates radiance into the volume. We originally implemented the photon marching portion of the code using the GPU's rendering pipeline, then re-implemented it using CUDA and achieved a 15- to 25-percent improvement in speed. For the photon storage part, we rely on the rendering pipeline to rasterize line segments into the volume texture representing the radiance distributions.

One technical limitation of this process is that current GPUs can only rasterize a line segment into a single 2D slice of a 3D volume texture. As a result, we are forced to take smaller steps than our octree would otherwise allow. This turns out to be a critical limitation, since photon tracing is the most expensive part of our rendering pipeline and its cost increases linearly with the number of steps in the photon paths. We therefore apply two strategies to mitigate the problem:

1. Instead of storing all the photon radiance in a 3D texture with a single slice orientation, we use a texture that has triple the number of slices and includes slices in all three orientations (i.e., three blocks of slices normal to the x, y, and z axes). When a photon takes a step, we choose which portion of the texture to rasterize the resulting line segment into based on the direction of the photon's motion; we use the slice orientation in which the photon can travel farthest before exiting a slice.

2. Second, we double the effective thickness of each slice by separating the volume texture into two render targets, one containing the even-numbered slices and the other containing odd-numbered slices (of all three orientations). When rasterizing a line segment, we render into an even and an odd slice simultaneously, using a fragment shader to draw or mask the pixels that belong in each.

In all, we end up using four render targets because we store radiance values and directions in separate textures and we split odd and even slices into separate textures. The added costs of having multiple slice orientations and multiple render targets are significantly outweighed by the speed-ups offered by longer photon step sizes. Note that we tried using eight render targets, but the performance decreases. The increased complexity of the fragment shader is one cause of the slowdown; we also suspect that we are taxing the texture-writing bandwidth of the graphics hardware.

We hope that the graphics pipeline will one day be capable of rendering line segments to volume textures without being limited to a single slice; this would negate the need for multiple slice orientations and multiple render targets. Alternatively, if CUDA offered an atomic read-modify-write operation for textures, we could implement our own line-segment voxelization code in CUDA without having to worry about multiple photons contributing radiance simultaneously to the same voxel. Relying on our own line voxelization in CUDA would eliminate the need for the large vertex buffer we currently use to transfer information from CUDA to OpenGL.

5.2 Graphics Pipeline or General Purpose?

There are several trade-offs to consider when choosing between using CUDA (or a similar framework) for general-purpose computation on the GPU and using OpenGL (or a similar API) to access the GPU's graphics pipeline. CUDA clearly offers a significant advantage to the developer when writing algorithms that don't divide nicely into a vertex shader, a geometry shader, and a fragment shader; it makes more sense to write general-purpose code than to massage the code into these artificially separated pieces. We can also avoid the costs associated with texture filtering and rasterization operations when we don't need them.

These advantages come at a cost, however, when mixing general-purpose computation with graphics-specific operations. In particular, roughly 10 percent of the time spent by our rendering pipeline is devoted to copying data back and forth between CUDA and OpenGL—currently, these technologies can share data in one-dimensional buffers but not 2D or 3D textures. We rely on CUDA to construct our octree, to generate photons from a shadow map, and to advance photons through the volume; we rely on OpenGL to voxelize objects, to create the shadow map, to rasterize photon contributions into the volume, and to integrate radiance along viewing rays. The interleaving of these steps requires that we copy the shadow map, the refractive index volume, and the extinction coefficient volume from OpenGL to CUDA. Fortunately, we can share the list of line segments generated by the CUDA implementation of photon marching with the OpenGL implementation of photon storage, but only because the list is a one-dimensional buffer. We look forward to a future in which CUDA and OpenGL can share two- and three-dimensional textures as well, in order to speed up our method and others that rely on both general-purpose computation and the graphics pipeline.

5.3 Modified Octree Construction

The appearance of caustics produced by our rendering pipeline is quite sensitive to the photon marching step size. The step size, in turn, is dependent on the tolerance ε used in our octree construction algorithm. For some volumetric data, we find that it's very difficult to choose a tolerance that yields an accurate rendering with a reasonable number of photon marching steps. In particular, we sometimes find that acceptable rendering quality demands such a low tolerance that the step sizes are too small to achieve interactive frame rates. A slight increase in the tolerance may cause many octree nodes to be merged, resulting in much larger step sizes and dramatically reduced rendering quality. We therefore modify our octree construction algorithm slightly to produce octree nodes with intermediate step sizes.

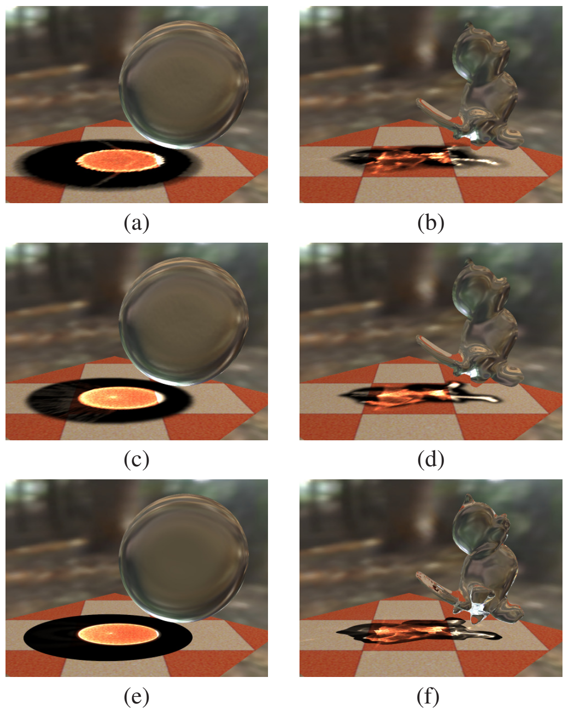


Figure 6: *Two objects with refractive index of 1.1, as rendered by (a, b) eikonal rendering; (c, d) our technique; and (e, f) ground-truth ray tracer.*

We amend our octree storage to hold in each voxel a maximum step size $\Delta s_{\max}$, in addition to the level number of the surrounding leaf-level octree node. We also make a slight change to the adaptive photon tracing algorithm in Section 4.4: we choose the step size Δs by clamping Δs_{octree} between the user-specified minimum step size $\Delta s_{\min}$ and the octree node's maximum step size $\Delta s_{\max}$.

We choose the maximum step size for each voxel during the octree construction stage. Whenever a voxel is assigned an octree level number because the variation in refractive index is smaller than ε, we set the voxel's step size limit $\Delta s_{\max}$ to infinity. If the variations in refractive index are larger than ε but smaller than a second user-specified tolerance ε', then we set the voxel's $\Delta s_{\max}$ to a finite step size limit chosen by the user. This scheme guarantees that within nodes of essentially constant refractive index, we advance photons all the way to the node boundary, while within nodes with some variation in refractive index, we limit the step size. In practice, we typically use a primary tolerance value of $\varepsilon = 0.005$, and associate an 8-voxel step size limit with a secondary tolerance of $\varepsilon' = 0.02$.

6 Results and Applications

To test our technique, we compared its output with that of a widely available off-line ray tracer and that of the eikonal renderer implemented by Ihrke et al. [2007]. We found that our method can more accurately reproduce the size and shape of caustics than the eikonal renderer, when compared to ground-truth renderings produced by a ray tracer. Figure 6 shows two examples of such comparisons.

There are at least two reasons for the greater accuracy of our technique. First, we accumulate the radiance contributions of all incoming photons within a voxel, while eikonal rendering only stores the contribution of the highest energy wavefront patch that passes through a voxel. As a result, eikonal rendering doesn't capture the caustics that should be generated by several low-intensity wavefronts converging from different directions—for example, the

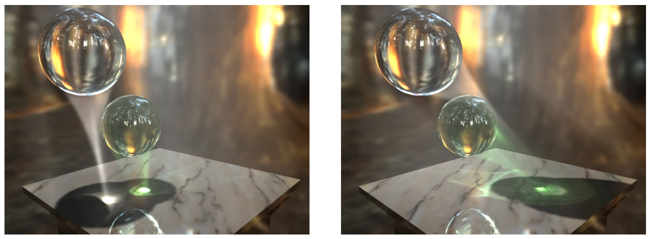

(a) (b)

Figure 7: *Renderings produced by our technique showing caustics before and after an interactive change to the light source position.*

bright spot in the middle of the sphere's caustic in Figure 6. Second, we are able to propagate photons through the volume without any difficulty as they pass through caustics, while eikonal rendering suffers from malformed wavefront quadrilaterals whenever the four corners of a patch pass through caustic singularities at different times. Some of these malformed patches are detected and eliminated, resulting in an artificial loss of radiance, while those that remain introduce numerical error visible as noisy, jagged edges in the caustics. As an object's index of refraction increases away from one, the wavefronts cross earlier and the propagation artifacts become increasingly visible. Photon tracing, on the other hand, produces acceptable results regardless of refractive index.

We also found that our approach is much faster than eikonal rendering at producing images after a change to the lighting or material parameters. We can relight scenes like those in Figure 6 at roughly eight frames per second, while eikonal rendering takes on the order of seven seconds for just one frame. The key to our technique's performance is its use of adaptive step sizes when tracing photons. By taking fewer steps through the volume, we generate fewer primitives to be rendered, and leverage the GPU's speed for rasterizing line segments instead of points into the radiance storage.

Table 1 demonstrates how our algorithm's performance depends on several parameters, including the number of triangles in the input mesh, the number of occupied voxels, and the number of initial photons. The input parameters being varied are shown in blue, and significant variations in the resulting statistics are shown in red. Because there is considerable overhead required for the transfer of data between OpenGL and CUDA, the timing is not simply proportional to the number of photon tracing steps. Four components of the total time are not shown here: gradient generation after voxelization, octree construction, irradiance merging after photon tracing, and the viewing pass. Each of these takes time proportional to the total resolution of the volume. The time required for the viewing pass is also linear in the number of output pixels.

number of triangles	occupied voxels	initial photons	millions of steps	voxeli-zation	photon tracing	total time
10,000	103^3	1024^2	3.81	14.55	136.85	225.20
20,000	103^3	1024^2	3.81	24.40	138.10	238.15
40,000	103^3	1024^2	3.81	49.90	134.70	258.40
80,000	103^3	1024^2	3.81	90.35	135.75	299.65
20,000	64^3	1024^2	1.12	16.40	63.55	148.00
20,000	77^3	1024^2	1.82	18.80	85.20	175.05
20,000	90^3	1024^2	2.72	21.45	108.60	201.75
20,000	103^3	1024^2	3.81	24.40	138.10	238.15
20,000	103^3	256^2	0.24	25.85	63.40	159.60
20,000	103^3	512^2	0.95	24.90	80.10	175.95
20,000	103^3	1024^2	3.81	24.40	138.10	238.15

Table 1: *Parameter variations (in blue) and their effects on performance (in red), for a scene that is otherwise fixed. The last three columns give times in milliseconds.*

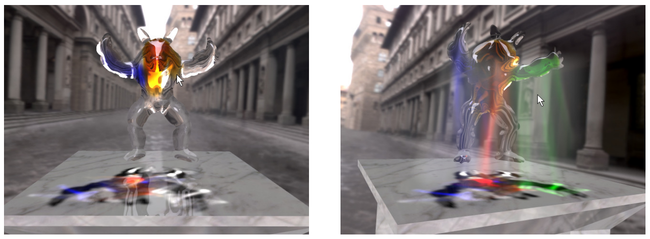

(a) (b)

Figure 8: *Painting with brushes that affect (a) absorption coefficients inside and (b) scattering coefficients outside an object.*

The speed of our approach opens the door to many novel applications. While previous rendering techniques for refractive media were capable of changing only the viewing parameters interactively, we can now permit changes to the lighting and material parameters as well, while still achieving interactive updates. We describe just a few of the possible applications below, and show a summary of their performance characteristics in Table 2.

6.1 Interactive Relighting

Our novel scheme of adaptive non-linear photon tracing allows us to achieve interactive performance of the rendering pipeline, even when it includes the lighting pass as well as the viewing pass. Users of our system can freely change the positions and colors of light sources while viewing the effects of caustics, absorption, single scattering, and shadows. Figure 7 illustrates the effect that a change in lighting produces in renderings of a refractive object, with volume caustics visible in the participating media.

6.2 Dynamic Materials

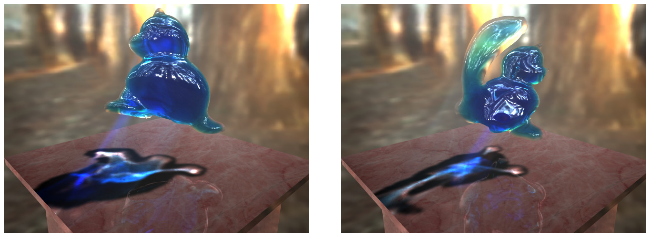
Our pipeline supports dynamic updates to material properties as well as lighting. The materials within a volume are defined by three properties: the index of refraction, the extinction coefficient, and the scattering coefficient. Variations in the index of refraction determine how light rays bend as they pass through the volume. The extinction coefficient affects how much light propagates through the volume from the light sources and to the camera. The scattering coefficient determines how much the radiance within each voxel contributes to the final image seen by the camera. There are many possible ways of interactively changing the material properties within a volume; we describe below three such techniques.

6.2.1 Volume Painting

One way of editing a volumetric description of materials is by "painting" directly into the volume. We can offer the user several "brushes," each consisting of a precomputed pattern of material parameters. We allow the user to drag a brush through the larger volume, and we add the appropriate scaled, rotated, and weighted patterns to the material properties of the volume. The example in Figure 8 shows some frames of a user's interactive painting session.

6.2.2 Surface Manipulation

While volume painting is handy for populating a volume with participating media, it's an unwieldy approach for creating a solid object or changing its shape. Our rendering pipeline includes a voxelization method capable of converting triangle meshes into volumetric data at interactive rates, and therefore we can integrate many familiar surface modeling algorithms into our system while still displaying the effects of refraction.



(a) (b)

Figure 9: *Volume and surface caustics produced by a refractive object as the user applies interactive surface deformations.*

We allow the user to interactively deform a refractive object and view the effect of the deformation on the scene's illumination. Because the combination of voxelization and rendering already taxes current GPUs, we rely on the CPU to perform the deformation. We apply the algorithm of Pauly et al. [2003] to deform a triangle-mesh surface according to the user's input, and perform our voxelization, octree construction, photon tracing, and viewing pass to display the result. Figure 9 shows some results of interactive deformations.

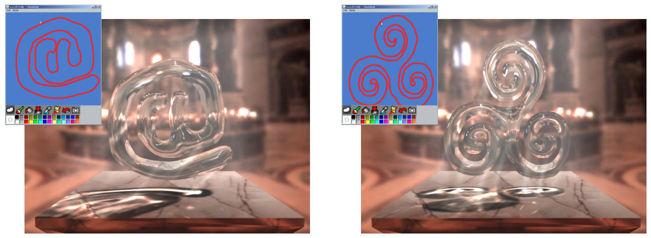
Sketch-based modeling has recently become quite popular because it is intuitive and convenient. As shown in Figure 10, we can integrate sketching into our system by using the method of Igarashi et al. [1999] to generate a surface mesh, which we then voxelize and send through our rendering pipeline. Indeed, any interactive surface modeling technique can supply input to our rendering system, since we have provided a general framework for voxelizing surface meshes and rendering the resulting refractive volumes.

6.2.3 Simulation and Animation

Material properties can be driven by physical simulations of real processes, or by animations of imagined ones. The wide range of literature on the simulation of solid and fluid dynamics provides a rich source of data for our rendering pipeline. For example, Figure 1 illustrates the use of our rendering technique on fluid simulation data from Mullen et al. [2007] and turbulent mixing of inhomogeneous materials from Fedkiw et al. [2001]. It also shows an example in which a keyframe-based animation of surfaces serves as input to our rendering pipeline. In each of these cases, the GPU-based pipeline can render arbitrary views of the changing material data under dynamic lighting conditions.

figure	number of triangles	photons propagated	millions of steps	frames per second
1 (a)	–	185,000	11.5	2.00
1 (b)	–	61,000	3.5	3.54
1 (c)	5,000	143,000	9.5	2.52
6 (c)	–	42,000	2.8	7.90
6 (d)	–	36,000	2.7	7.88
7	–	68,000	3.4	6.14
8	–	41,000	2.0	7.24
9	20,000	35,000	1.7	6.22
10 (a)	2,900	75,000	3.5	5.43
10 (b)	5,000	70,000	3.2	5.59

Table 2: *Performance of our technique. Each example image was rendered at 640 × 480 resolution using an Nvidia GeForce 8800 Ultra, starting with 1024 × 1024 photons and a 128 × 128 × 128 volume. Included are the number of triangles in the input mesh (where applicable), the number of photons traced through the volume, the number of photon-tracing steps, and average frame rate.*



(a) (b)

Figure 10: *Two examples of refractive objects created using an interactive sketching tool.*

7 Conclusion

We have presented a technique capable of rendering the effects of refraction, single scattering, and absorption at interactive rates, even as the lighting, material properties, geometry, and viewing parameters are changing. This is a significant improvement over previous techniques, which have not been capable of simultaneous interactive updates to lighting, materials, and geometry. Our rendering pipeline leverages the GPU to convert surface meshes to volumetric data, to trace photons through the scene, and to render images of the resulting radiance distribution. We leave the CPU free for applications such as interactive surface deformation and the simulation of physical processes.

Although our approach is similar to volume photon mapping, we rely on a regular voxel grid rather than a sparse kd-tree to store the distribution of radiance within the volume. While one could argue that a voxel grid does not scale well to large or complex scenes, we claim that for many scenarios involving a single refractive object or a confined participating medium, a voxel grid is adequate. Furthermore, by using a voxel grid, we can take advantage of the GPU's rasterization pipeline to store photon radiance contributions during the lighting pass, and we avoid the hefty cost of searching for nearby photons during each step of the viewing pass.

Our technique also shares some similarities with eikonal rendering, but achieves greater speeds, primarily by using adaptive step sizes. We are free to change the step size because we propagate individual photons rather than a coherent wavefront. In addition, we avoid the difficulties encountered in eikonal rendering when wavefront patches become inverted as they pass through caustic singularities.

Our rendering pipeline relies on both OpenGL and CUDA for its implementation, and therefore we hope to see broader support for applications that mix general-purpose computing and graphics-specific operations on the GPU. In particular, our approach could be further accelerated if OpenGL were able to render primitives into more than one slice of a 3D texture, if CUDA were able to read, modify, and write a texture entry in a single atomic operation, or if 2D and 3D textures could be shared without copying.

The amount of video memory on current graphics hardware limits our approach to volume resolutions of 128 × 128 × 128. As a result, we cannot render very complex scenes, and surface caustics and shadows become blurred. Temporal coherence can also be an issue, since lighting changes may result in slightly different photon storage patterns in a low-resolution volume. The incorporation of volumetric approaches like our method and eikonal rendering into traditional polygon-based renderers is an open research problem.

We envision several areas for future work. Accounting for multiple scattering during the adaptive photon tracing process would allow us to render fog, smoke, and steam more realistically. We are also interested in incorporating physical simulations into our system, particularly for phenomena like fluids and mirages, where the effects of refraction are significant. Because our rendering pipeline already taxes the GPU, we face challenges of balancing the work between the central processor and the graphics processor, and of speeding the transfer of volumetric data between the two. Finally, we would like to go beyond point and directional light sources to see how environment maps and other complex light sources can be incorporated into our rendering pipeline.

Acknowledgements

We would like to thank the reviewers for their valuable comments, and Linjie Luo for help in turbulence generation. Xin Sun and Jiaoying Shi were partially supported by the 973 Program of China (No. 2002CB312105). This work was done while Xin Sun was an intern at Microsoft Research Asia.

References

BORN, M., AND WOLF, E. 1999. *Principles of Optics (7th edition)*. Cambridge University Press.

CHATTERJEE, S., BLELLOCH, G. E., AND ZAGHA, M. 1990. Scan Primitives for Vector Computers. In *Supercomputing 1990*, 666–675.

CRANE, K., LLAMAS, I., AND TARIQ, S. 2007. Real-Time Simulation and Rendering of 3D Fluids. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, ch. 30, 633–675.

DAVIS, S. T., AND WYMAN, C. 2007. Interactive Refractions with Total Internal Reflection. In *Graphics Interface 2007*, 185–190.

DOBASHI, Y., YAMAMOTO, T., AND NISHITA, T. 2002. Interactive Rendering of Atmospheric Scattering Effects Using Graphics Hardware. In *Graphics Hardware 2002*, 99–107.

EISEMANN, E., AND DÉCORET, X. 2006. Fast Scene Voxelization and Applications. In *Proceedings of I3D 2006*, 71–78.

ERNST, M., AKENINE-MÖLLER, T., AND JENSEN, H. W. 2005. Interactive Rendering of Caustics Using Interpolated Warped Volumes. In *Graphics Interface 2005*, 87–96.

FANG, S., AND CHEN, H. 2000. Hardware Accelerated Voxelization. *Computers and Graphics 24*, 3, 433–442.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual Simulation of Smoke. In *Proceedings of ACM SIGGRAPH 2001*, 15–22.

GUTIERREZ, D., SERON, F. J., ANSON, O., AND MUÑOZ, A. 2004. Chasing the Green Flash: A Global Illumination Solution for Inhomogeneous Media. In *Spring Conference on Computer Graphics 2004*, 97–105.

GUTIERREZ, D., MUNOZ, A., ANSON, O., AND SERON, F. J. 2005. Non-linear Volume Photon Mapping. In *Rendering Techniques 2005*, 291–300.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A Sketching Interface for 3D Freeform Design. In *Proceedings of ACM SIGGRAPH 99*, 409–416.

IHRKE, I., ZIEGLER, G., TEVS, A., THEOBALT, C., MAGNOR, M., AND SEIDEL, H.-P. 2007. Eikonal Rendering: Efficient Light Transport in Refractive Objects. *ACM Trans. Graph. 26*, 3, 59:1–59:9.

IWASAKI, K., DOBASHI, Y., AND NISHITA, T. 2002. An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware. *Computer Graphics Forum 21*, 4, 701–711.

JENSEN, H. W., AND CHRISTENSEN, P. H. 1998. Efficient Simulation of Light Transport in Scenes with Participating Media Using Photon Maps. In *Proceedings of ACM SIGGRAPH 98*, 311–320.

JENSEN, H. W. 1996. Global Illumination Using Photon Maps. In *Rendering Techniques 96*, 21–30.

KAJIYA, J. T., AND HERZEN, B. P. V. 1984. Ray tracing volume densities. In *Proceedings of ACM SIGGRAPH 84*, 165–174.

KRÜGER, J., BÜRGER, K., AND WESTERMANN, R. 2006. Interactive Screen-Space Accurate Photon Tracing on GPUs. In *Rendering Techniques 2006*, 319–329.

MOON, J. T., WALTER, B., AND MARSCHNER, S. R. 2007. Rendering Discrete Random Media Using Precomputed Scattering Solutions. In *Rendering Techniques 2007*, 231–242.

MULLEN, P., MCKENZIE, A., TONG, Y., AND DESBRUN, M. 2007. A Variational Approach to Eulerian Geometry Processing. *ACM Trans. Graph. 26*, 3, 66:1–66:10.

NVIDIA CORPORATION. 2007. CUDA Programming Guide. http://developer.nvidia.com/object/cuda.html.

OLIVEIRA, M. M., AND BRAUWERS, M. 2007. Real-Time Refraction Through Deformable Objects. In *Proceedings of I3D 2007*, 89–96.

PAULY, M., KEISER, R., KOBBELT, L. P., AND GROSS, M. 2003. Shape Modeling With Point-Sampled Geometry. *ACM Trans. Graph. 22*, 3, 641–650.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon Mapping on Programmable Graphics Hardware. In *Graphics Hardware 2003*, 41–50.

STAM, J., AND LANGUÉNOU, E. 1996. Ray Tracing in Non-Constant Media. In *Rendering Techniques 96*, 225–234.

SUN, B., RAMAMOORTHI, R., NARASIMHAN, S. G., AND NAYAR, S. K. 2005. A Practical Analytic Single Scattering Model for Real Time Rendering. *ACM Trans. Graph. 24*, 3, 1040–1049.

SZIRMAY-KALOS, L., ASZÓDI, B., LAZÁNYI, I., AND PREMECZ, M. 2005. Approximate Ray-Tracing on the GPU with Distance Impostors. *Computer Graphics Forum 24*, 3, 695–704.

WEISKOPF, D., SCHAFHITZEL, T., AND ERTL, T. 2004. GPU-Based Nonlinear Ray Tracing. *Computer Graphics Forum 23*, 3, 625–633.

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6, 343–349.

WYMAN, C., AND DAVIS, S. 2006. Interactive Image-space Techniques for Approximating Caustics. In *Proceedings of I3D 2006*, 153–160.

WYMAN, C. 2005. An Approximate Image-space Approach for Interactive Refraction. *ACM Trans. Graph. 24*, 3, 1050–1053.

ZHANG, L., CHEN, W., EBERT, D. S., AND PENG, Q. 2007. Conservative Voxelization. *Visual Computer 23*, 9, 783–792.

ZIEGLER, G., DIMITROV, R., THEOBALT, C., AND SEIDEL, H.-P. 2007. Real-Time Quadtree Analysis Using HistoPyramids. In *Real-Time Image Processing 2007*, vol. 6496.