APPROVAL SHEET

Title of Thesis: Real-Time Multiple Refractions Through Deformable Objects

Name of Candidate: Pankaj Purushottam Chaudhari
M.S. in Computer Science, 2008

Thesis and Abstract Approved: _____

Dr. Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: Pankaj Purushottam Chaudhari.

Permanent Address: 4709 Belwood Green, Baltimore, MD 21227.

Degree and date to be conferred: M.S. in Computer Science, August 2008.

Date of Birth: 08-23-1982.

Place of Birth: Jalgaon, India.

Secondary Education: Laxmanrao Apte Prashala Junior College, Pune, India.

Collegiate institutions attended:

University of Maryland Baltimore County, M.S Computer Science, 2008.
Pune Institute of Computer Technology, B.E. Computer Engineering, 2004.

Major: Computer Science.

Professional positions held:

Member of Technical Staff, Great Software Laboratory Pvt. Ltd., Pune, India. (September 2005 – July 2006).

Assistant System Engineer (Trainee), Tata Consultancy Services Ltd., Mumbai, India. (August 2004 – August 2005).

ABSTRACT

Title of Thesis: Real-Time Multiple Refractions Through Deformable Objects

Pankaj Purushottam Chaudhari, M.S. in Computer Science, 2008

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

Refraction of light is one of the important phenomena that contribute to the perception of realism and is responsible for widely observed effects such as caustics. We describe an image-space algorithm to simulate refraction of light through a dynamic object. Unlike previous approaches, our technique achieves refraction through multiple interfaces at interactive frame-rates and does not require any pre-processing. Each stage of our algorithm runs entirely on the GPU and fits into the existing raster-based pipeline. Multiple refractions are simulated using depth layers obtained from a dynamically voxelized polygonal model and by using an image-space ray tracing technique. We also describe an accurate surface interpolation technique used to obtain surface normals using the depth layers. Our implementation suits the standard rasterization pipeline and achieves visually plausible results at interactive frame rates even for the dynamic scenes involving multiple refractive objects.

Real-Time Multiple Refractions Through Deformable Objects

by

Pankaj Purushottam Chaudhari

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
M.S. in Computer Science
2008

© Copyright by

Pankaj Purushottam Chaudhari

2008

Dedicated to the inventors of Café Mocha.

ACKNOWLEDGMENTS

I thank my advisor Dr. Marc Olano for his support and guidance throughout this thesis work. I also want to thank my committee members for reviewing this document and providing valuable feedback on the text. Thanks to my parents for their never-ending endurance with me during the tough times.

I am pleased to thank all VANGOGH Lab members for their time and support. I am grateful to my roommates for adjusting with my nocturnal schedule. My special thanks to my friend Varsha Rao for untiringly reviewing many drafts of this document.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

From the time of early civilizations, philosophers and researchers alike have tried to explain the intriguing properties of light. Ancient Greek and Indian thinkers included light amongst the five fundamental elements which make matter. In conjunction, they also believed that light was an atomic entity equivalent to energy. In modern physics, several theories about the nature of light have been proposed. According to the well-known *particle theory*, light is made up of tiny particles, called *photons*, which are emitted from a light source. Photons are the carriers of electromagnetic waves of different wavelengths. They possess zero mass and travel at extremely high velocity. Our brain perceives an image of the world based on the properties of these photons. Our perception of the color of an object is simply the brain's interpretation of the density of photons of a particular wavelength, traveling from an object, falling on our retina.

The propagation of light has been studied extensively and influence of the medium of propagation on the properties of light had not escaped the focus of scientists. During the time of the Scientific Revolution, the findings of several scholars have added to our knowledge on the propagation of light. A classic example being the property of light to disperse into its seven component colors discovered by Newton. According to the modern particle theory, the photons that comprise light can undergo *absorption*, *emission*, *reflec-*

tion, or *transmission* when they collide with the objects in their path. This behavior of photons is responsible for many optical phenomena such as shadows, reflection, refraction, dispersion, caustics and fog. Computer graphics involves simulation of such behavior using computers. In reality, the immense speed of photons generates optical effects in no time. However, due to limited computational power, the problem of simulating complex optical phenomena even for a simple object is still an interesting challenge to researchers.

Refraction of light is one of the important phenomena that greatly contribute to the perception of realism. Refraction occurs whenever light enters a transparent medium where its speed is different. This difference in speed causes bending of light rays as shown in Figure 1.1. Curved transparent surfaces often cause bent light rays to converge on the surrounding diffuse objects forming bright patterns of light known as *caustics*. Realism of refraction can often be visualized using its caustics pattern. Figure 1.1 shows an example of refractive caustics formed by a drinking glass onto a table. Caustics can also be formed by the light rays reflected by curved specular surfaces.

FIG. 1.1. Left image shows real-life refraction caused by water and the right image shows an example of refractive caustics produced by a drinking glass on a table.

Ray tracing can accurately simulate refraction. However, due to the recursive nature

of refraction, doing it in real-time still remains a challenge. High computational and storage complexity of these problems often leads to approximation of realistic parts. Many interactive applications require such realism to be produced in real time. A few examples include 3D games, virtual reality applications, and 3D mesh modeling applications for the artists requiring immediate visual feedback.

Researchers have come up with image-space approaches by exploiting the capabilities of modern graphics hardware to approximate refraction through two surfaces (Wyman 2005a; Oliveira & Brauwers 2007). Refraction through multiple surfaces was previously achieved using layered depth images on static models (Krüger, Bürger, & Westermann 2006). In addition, Eikonal rendering has been used to achieve the same in real-time but by using volumetric representation of inhomogeneous static scenes (Ihrke *et al.* 2007). In this thesis, we present an interactive framework to simulate transmission of photons, that is refraction of light, through complex transparent objects with dynamically changing shapes.

Our work is inspired by the work of Oliveira et al. (2007) and Kruger et al. (2006). Contrary to previous multiple surface refraction techniques (Ihrke *et al.* 2007), it requires no preprocessing. Our technique employs surface voxelization of dynamic models to obtain surface occupancy information in a uniform grid stored as a 2D texture. Using a high-resolution volumetric representation allows us to obtain up to 32 depth layers of any dynamic object represented as a polygonal mesh. Light rays are intersected against these surfaces using a fast binary search algorithm. At each intersection found, we extract surface information by interpolating the surface normal using the depth gradient from the depth image generated using the voxel occupancy information. Since our method fits completely into the existing rasterization-based rendering pipeline, it can be combined with image-space caustics visualization technique such as caustics mapping. Interactive performance of our method to render dynamic objects makes it suitable for real-time applications such as video games that require visual realism and involve animated or deforming models.

The main contributions of this thesis include:

- A real-time method to obtain up to 32 depth layers by voxelizing a dynamic object;
- A perspective-corrected depth gradient method to estimate surface normals using these depth layers;
- An adaptive image-space binary search algorithm to incorporate refraction through multiple layers.

Chapter 2

BACKGROUND AND RELATED WORK

In this section, we discuss related work in this field and also explain a few background concepts needed to understand this thesis.

2.1 Refraction and Caustics

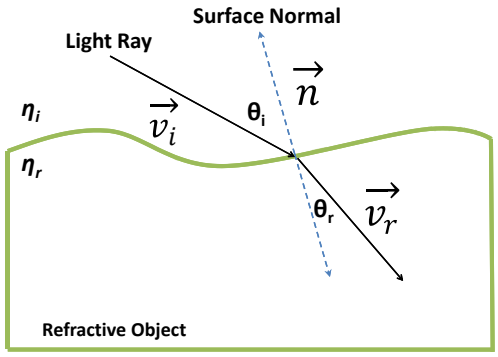


FIG. 2.1. Illustration Snell's law of refraction to model bending of light rays.

Bending of light rays can be easily modeled by applying Snell's law at each point of refraction. Snell's law defines the relationship between the angles of incidence and

refraction as follows:

$$\eta_i * sin(\theta_i) = \eta_r * sin(\theta_r),$$

Where,

η_i is the refractive index of the medium the light is leaving. The refractive index of a medium is simply the ratio of the speed of light in a vacuum to the speed of light in that medium,

η_r is the refractive index of the medium the light is entering,

θ_i is the incident angle, the angle made by the entering light ray, $\vec{v_i}$, with the surface normal, $\vec{n}$, at the point of refraction,

θ_r is the angle of refraction, the angle made by the leaving light ray, $\vec{v_r}$, with the surface normal, $\vec{n}$, at the point of refraction.

For a complex object in which a light ray gets refracted at multiple interfaces, Snell's law can be applied successively at each interface. Caustics are formed when these refracted rays exit the object and hit the surrounding diffuse environment. Caustics are the boundaries where the intensity of light changes from zero to non-zero, that is, from shadow to bright pattern of light. Increased intensity at a point occurs due to multiple photons hitting it. Thus caustics can be easily modeled as an accumulation of intensity of the photons when they hit the diffuse surrounding.

2.2 Ray Tracing And Photon Mapping

The most obvious way to propagate light rays through any scene is to use a conventional ray tracing approach that involves following the path that light takes as it bounces through an environment by subsequent recursive application of Snell's law. As shown in Figure 2.2, a light ray is shot for each pixel on the screen. This light ray is then intersected with the objects in the scene and shading calculations are performed at the closest point of

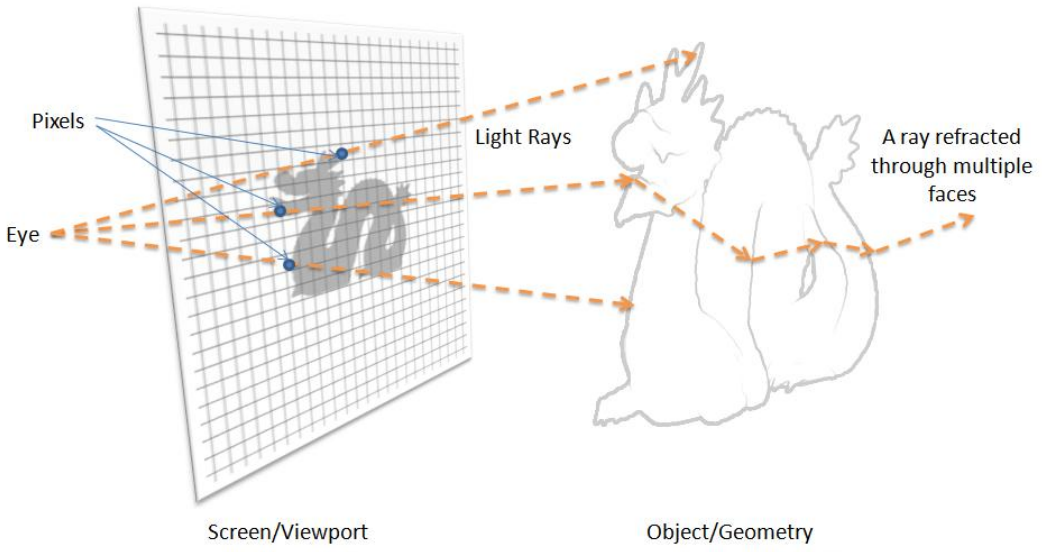
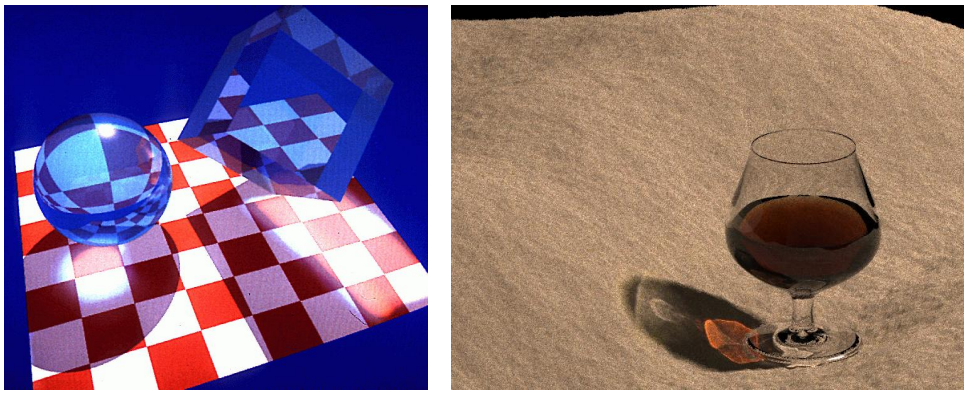


FIG. 2.2. Illustration of how ray tracing works. A light ray from each pixel is intersected with the objects in the scene. These rays can be reflected or refracted recursively at each intersection. Here, the middle ray is refracted through multiple interfaces.

intersection. At each intersection point, light rays are reflected or refracted depending upon the object's optical properties. This approach is not suitable to render caustics, as light rays must be cast from the light source instead of the eye. Ray tracing from the light source is known as *photon tracing*.

The problem of rendering caustics is closely related to the problem of rendering refraction. However, it requires storing the light energy, that is the luminosity, at each point in three-dimensional space. Arvo (1986) introduced illumination maps to store light energy in a pre-processing step using *photon tracing*. During the rendering pass, ray tracing from the eye was then used to illuminate the scene by utilizing these illumination maps in a shader. Figure 2.3(a) shows one of the first synthetic images demonstrating caustics produced using this approach. Although ray tracing produces realistic results, it is not suitable for real-time applications. In addition, performing independent operations at each pixel is not suitable for the streaming architecture of modern graphics hardware.



(a) (b)

FIG. 2.3. Arvo (1986) introduced photon tracing approach to render caustics (a). A glass of cognac rendered using photon mapping (b) from Jensen (1996).

Jensen (1996) introduced a two-pass method to accomplish global illumination using photon maps (Figure 2.3(b)). Photon maps are generated by emitting photons from the light sources and storing photon locations as they hit surfaces. Photon hits are stored in a hierarchical tree structure to facilitate fast searching of the nearest photons hits to estimate radiance of any point in the scene. Photon mapping can provide full but non-interactive global illumination solution. There have been advances in parallelizing photon mapping (Günther, Wald, & Slusallek 2004) onto a cluster of PCs to obtain interactive caustics; however, using a cluster of PCs is not suitable for everyday real-time applications like games.

2.3 Interactive refraction using Graphics Processing Units (GPUs)

Several approaches have been proposed to achieve simulation of refraction on GPUs. This section will describe the rendering pipeline found in modern GPUs and previous efforts applied to obtain interactive refraction and caustics.

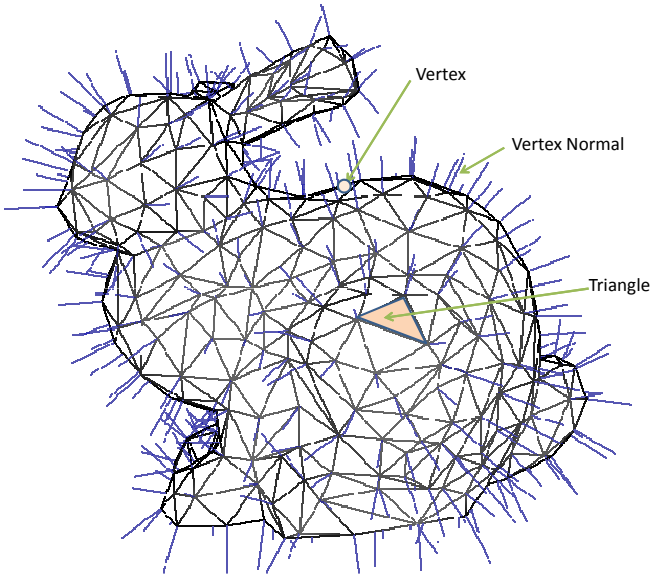


FIG. 2.4. On a GPU, an object is represented as a mesh of triangles. Each triangle has three vertices and a vertex normal at each of them specifies the orientation of a triangle.

2.3.1 GPU pipeline

On a GPU, an object is represented as a list of primitives, mainly points, lines or triangles. More commonly, a triangular mesh (Figure 2.4) is utilized to describe the shape of an object. A triangle is described by three points, known as *vertices*, and their normals, known as *per-vertex normals*. Vertices are normally defined in the object's own coordinate space or *object space*. To render an object on a 2D screen, several transformations are required to be performed on its vertices. In a typical scene, an object can be placed anywhere by transforming these vertices from the object's space to the world-space. These world coordinates are then transformed to *camera-space* and then mapped to 2D screen pixels using the camera's projection parameters.

Currently available graphics processors provide parallel processing capability using multiple stream processors that are capable of performing custom but uniform operations on streamed data such as vertices, geometry and pixels. Separate custom operations, known

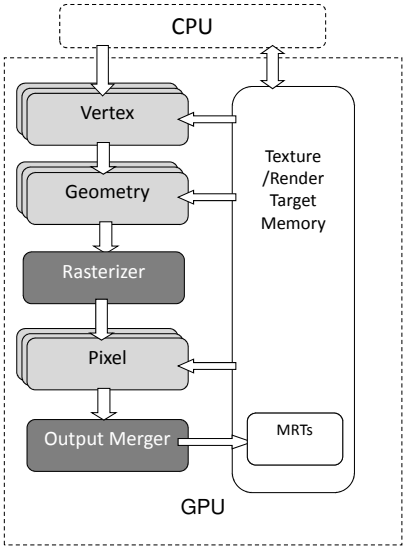


FIG. 2.5. Rendering pipeline in modern GPUs. Vertex, geometry and pixel stages are programmable; however, Output Merger stage has a fixed functionality. Output can be written to multiple render targets (MRTs).

as shaders, can be written for vertices, geometry and pixels. Figure 2.5 shows important stages in the GPU pipeline. *Vertex shaders* are generally responsible for transforming the vertices of each primitive from object-space to screen-space and assigning per-vertex attributes, such as color. *Geometry shaders* are executed per-primitive to emit or destroy primitives. A primitive can be a point, a line or a triangle. Primitives are then rasterized by a fixed function stage, known as a *rasterizer*. Rasterization of a 3D primitive involves mapping it onto a 2D screen by finding potential pixels, or *fragments*, that it occupies. A rasterizer is also responsible for interpolating the surface normal for each pixel using the vertex normals of a primitive. These potential pixels are then processed by *pixel shaders* running in parallel to determine final color of a pixel. Output colors can be blended together using a configurable stage called the *output merger*. In Direct3D, the output merger can be configured for additive, subtractive or min-max blending. However, OpenGL provides many other blending functions including logical operations. Output colors can be written directly

to the screen or to intermediate textures called *render targets* allowing for multiple-pass lighting effects. Modern GPUs provide the capability to render to up to eight simultaneous render targets. More recently, the Compute Unified Device Architecture (CUDA) has been introduced on specific graphics hardware to facilitate general purpose floating point computations on the GPUs using a *c*-like language for parallel processing.

2.3.2 Image-space refraction

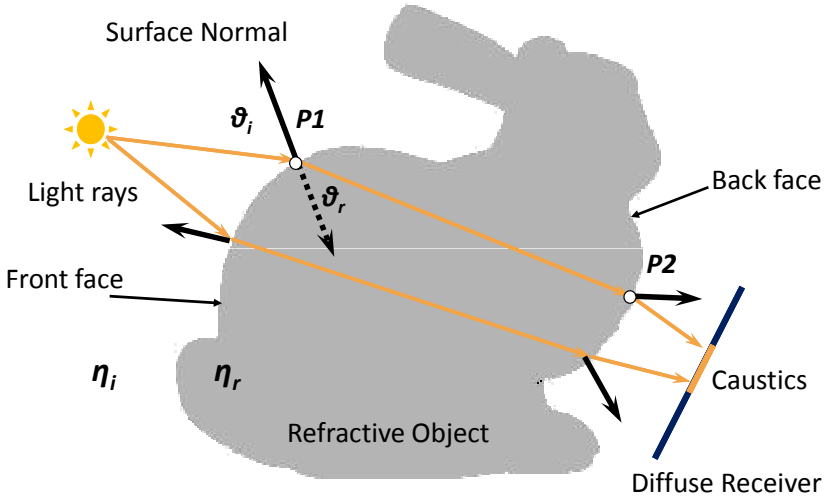


FIG. 2.6. Illustration of interaction of light rays with front and back faces of an object. After rasterization, the information about point $P1$, normal $\vec{n}$ and incident vector $\vec{v_i}$ is available. However, depth testing discards any information about exit point $P2$ and its corresponding normal.

Several approaches have been proposed by researchers to approximate refraction and caustics on a graphics processing unit (GPU). Focusing on real-time rendering, Wyman

(2005a) introduced an *image-space* method for approximating refraction through two interfaces of static objects. The ray tracing algorithm described in the previous section finds intersections of the light rays against the actual geometry of the objects. Such algorithms are classified as *object-space* algorithms. However, image-space algorithms operate on 2D images of an object rendered from a single point of view rather than on its actual geometry. Thus, such algorithms are not limited by the number of vertices or polygons in the scene. Rasterization is the stage which converts a geometry from the object-space to the image-space. Images utilized by such algorithms generally store per-pixel depth or surface normals. In an image-space approach, a fixed set of operations is performed for each pixel of a 2D image. It is thus suitable to the streaming architecture of the current GPUs which can perform per-pixel operations in parallel with great speeds. Since image-space algorithms work only on the visible pixels of a scene, they are fast and inherently provide better sampling. Furthermore, even for high resolution output images, per-pixel processing does not become a bottleneck.

In the image-space, the following information is easily available for each pixel after rasterization, (refer to Figure 2.6):

- incident light vector, $\vec{v_i}$,
- point of refraction, $P1$,
- surface normal at the point of refraction, $\vec{n}$

Using this information, the refracted ray, $\vec{v_r}$, can be easily computed using Snell's law. However, a pixel shader is not given any information about the point $P2$ from the backmost interface and its surface normal. This problem is solved by using a separate rendering pass to store depths and surface normals of the backmost faces in intermediate textures as shown in Figure 2.7. The depth of the object along each vertex normal is pre-computed and then

used in a pixel shader to approximate the location of point $P2$. Since for deforming objects depth along vertex normals constantly changes, this approach is not suitable to render them.

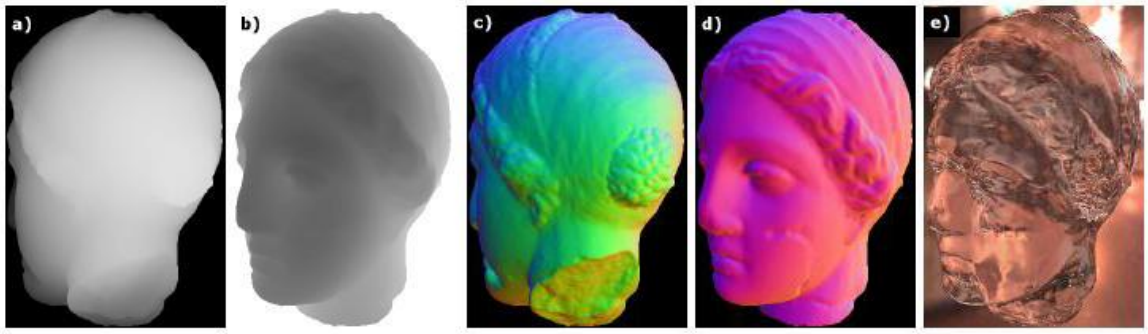


FIG. 2.7. Wyman (2005) used a separate pass to obtain depth and normal maps of the backmost surfaces (a) and (c). (b) and (d) are depth and normal maps for front faces. In a pixel shader, these textures are used together to get the final result (e).

Shah et al. (2007) adopted Wyman's (2005a) method for refraction and introduced a caustics mapping technique similar to shadow mapping. As in the case of backward ray tracing, the refractive object is rendered from the light source point of view to obtain exiting refracted rays or photons using Wyman's approach. These photons are then intersected with the surroundings in the pixel shader and corresponding 3D positions are stored in a texture. Point primitives are then rendered by using this position map as a vertex buffer. Caustics maps are created by splatting the point primitives and accumulating luminosity at each pixel. In the final pass of rendering from the eye, caustics maps are used to fetch luminosity information at each pixel. Wyman et al. (2006) observed that such an approach leads to noisy and incoherent images, and they thus used a nearby neighbor photon gathering algorithm in a 7x7 window along with Gaussian filtering. They also achieved coherency by storing photons for three recent frames. Recently, Wyman (2008) further accelerated the generation of the caustics map by eliminating unnecessary photons at the earliest point using hierarchical caustics maps. Figure 2.8 shows caustics rendered using these methods.

FIG. 2.8. Refraction and caustics produced by Wyman (left) and Shah (right).

Recently, Oliveira et al. (2007) introduced an image-space ray tracing approach to estimate the intersections of light rays with the backmost interface of deformable objects without any pre-computations. Point $P2$ (Figure 2.6) where the ray exits the object is computed by performing a binary search along the refracted ray $\vec{v_r}$. The search is optimized by restricting it by the bounds of the object in the viewing direction. This method can easily be used along with any of the previously discussed methods to render caustics. All the image-space techniques discussed so far are limited to modeling refractions only through front and back interfaces of the objects. Another group of researchers has developed refraction on complex geometry using spherical harmonics to store and retrieve ray-traced refraction paths for each vertex and then interpolate the same for each fragment during final rendering (Génevaux, Larue, & Dischler 2006). While this approach produces results closer to realism, it is nevertheless, very memory and computationally intensive. It also requires a long pre-computational step of generating and compressing spherical harmonics coefficients. Furthermore, reduced sampling while compressing using spherical harmonics introduces aliasing artifacts.

2.4 Volumetric Caustics

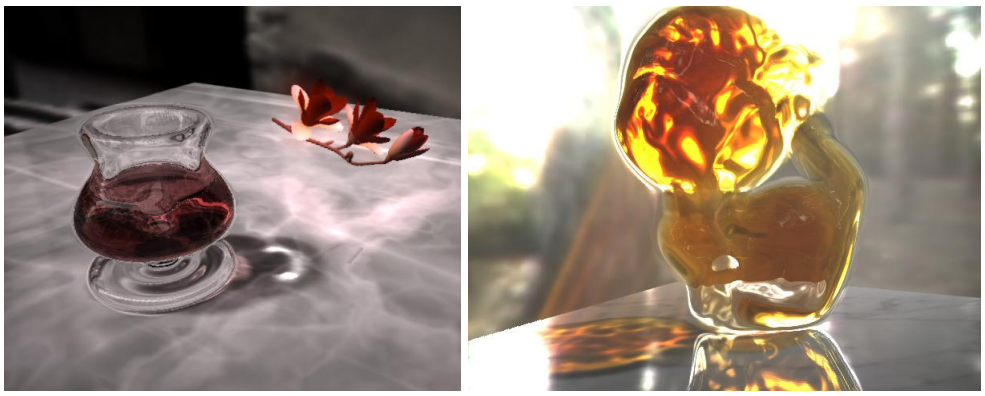


FIG. 2.9. Volumetric caustics produced by Ihrke et al. (2007) (left) and Sun et al. (2008) (right). Both the methods achieve effects such as volumetric caustics, absorption, scattering, etc. in real-time.

All methods described in the previous section are based on applying Snell's law of refraction at each interfering interface. Alternatively, volumetric representation of an object can be used for simulation of refraction. A high resolution volumetric data grid storing surface normals at each grid location can provide accurate information about surface details. However, it becomes highly memory inefficient to store normals for a large volume. Furthermore, to obtain such volumetric information for a deforming object is computationally expensive and thus not suitable for real-time rendering. Ihrke et al. (2007) take another approach and present a sophisticated method, *Eikonal Rendering*, to model refraction through volumes using geometric optics. They employ a light propagation scheme derived from the ray equation of geometric optics to estimate non-linear (curved) light paths through a volume with smoothly varying refractive indices. The following equation defines the path $\vec{x}$ of a photon in a field n of inhomogeneous refractive indices (Ihrke *et al.* 2007) with an infinitesimally small step size ds along the direction of light ray:

$$\frac{d}{ds}(n\frac{d\vec{x}}{ds}) = \vec{\nabla}n \tag{2.1}$$

where, $\vec{\nabla}n$ is the gradient of refractive indices along the ray direction. Using $\vec{w} = n\frac{d\vec{x}}{ds}$, the equation 2.1 can be rewritten as:

$$\frac{d\vec{x}}{ds} = \frac{\vec{w}}{n} \tag{2.2}$$

$$\frac{d\vec{w}}{ds} = \vec{\nabla}n \tag{2.3}$$

Using Euler forward-difference discretization of these continuous equations, a photon can be marched as follows:

$$\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta s}{n}\vec{w}_i \tag{2.4}$$

$$\vec{w}_{i+1} = \vec{w}_i + \Delta s\vec{\nabla}n \tag{2.5}$$

where, Δs is a discrete step size to advance photons along the curved path. Gradients of refractive indices at each grid location of the volume are obtained as an offline pre-processing step and stored in a 3D texture. In the rendering pass, photons are then traced through this volume of gradients using a constant step size as described in the Equations 2.4 and 2.5. Figure 2.9 shows many volumetric effects such as refraction, caustics, multiple scattering, attenuation and emission produced with this method in real-time. However, any change to the geometry in the scene requires recalculation of the gradients of refractive indices which takes several seconds and hence limits this approach only to static scenes.

Independently and concomitant with our work, Sun et al. (2008) conducted research

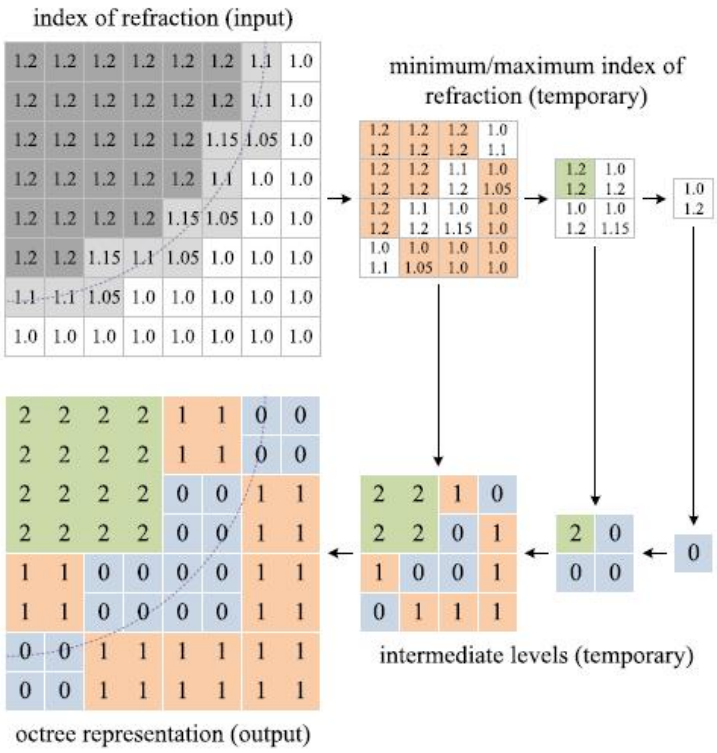


FIG. 2.10. Spatially varying refractive indices and the octree (shown in 2D) used by Sun et al. (2008). Value in the each cell of the octree indicates the step size to be taken to advance photons in the region nearby.

on overcoming such pre-computations by dynamically obtaining spatially varying refractive indices using voxelization of an object. They accomplished it by assuming that the refractive index varies only near the surfaces and is constant inside the object (Figure 2.10). The refractive index for a voxel was derived from the density of the object in it. Voxelization was performed by rasterizing the model giving only the refractive indices at the surfaces. The same equations 2.4 and 2.5 as in Eikonal rendering are used for photon tracing, however, with an adaptive step size. As shown in Figure 2.10, an octree storing the minimum and maximum of the refractive indices at each cell is generated in a separate pass. This octree is then utilized to vary the step size depending upon the gradient of refractive indices. For example, a larger step size can be taken for the region inside the object, where

the refractive index is almost constant. As seen in Figure 2.9, effects similar to Eikonal rendering can be produced. Although interactive, use of 3D textures limits this approach by the amount of video memory available. Furthermore, CUDA is used for photon tracing which restricts its use on specific graphics hardware.

It can easily be observed that if a scene consists of refractive objects with uniform refractive indices, then non-linear viewing ray propagation is unnecessary and Snell's law can simply be applied at each interfering interface of the objects. With this observation, the problem of multiple refractions simply reduces to performing linear ray tracing through the scene and applying Snell's law at all surfaces that come along the light path.

2.5 Voxelization using GPUs

Voxels are three dimensional entities representing volumetric information such as density, color, surface normal or simply occupancy, that is *in-out* information indicating whether some part of the object is inside a voxel or not. To voxelize a polygonal mesh, a grid of cells is constructed around it. For each polygonal primitive, the cells that it intersects are found. Figure 2.11 depicts a voxelized Buddha model.

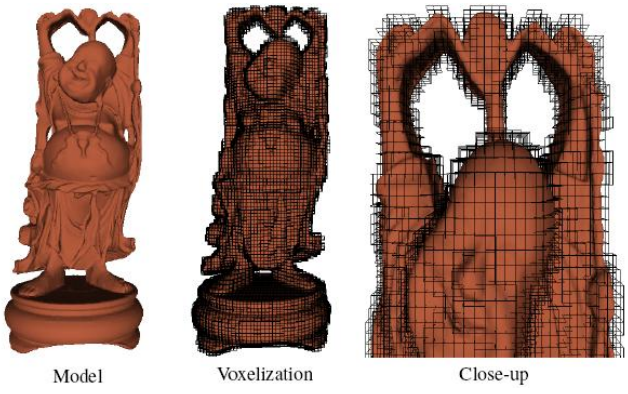


FIG. 2.11. Triangular Buddha model is voxelized by Varadhan et al. (2006). The model is divided into a uniform grid of cuboids known as *voxels*.

It is apparent that a technique is needed to obtain the volumetric representation of the refractive object in a single rendering pass. While the exact surface information cannot be obtained, volume occupancy information can easily be generated by real-time voxelization of polygonal models using GPUs. Dong et al. (2004) showed a possibility of achieving interactive voxelization for dynamic scenes. Eiseman et al. (2006) further accelerated voxelization by utilizing the GPU's blending functionality and stored surface occupancy information to a 2D texture. They also demonstrated its use in generating transmittance shadow maps and crude refraction in real-time. Crane et al. (2007) obtained an *in-out* volumetric representation in a 3D texture using a stencil buffer for orthogonal voxelization. This approach requires storing a vertex per grid cell, and hence becomes memory inefficient to generate large volumes. For lighting any refractive object, it is sufficient to know only about surface occupancy, as the index of refraction remains constant inside the object and scattering happens only at the interfering surfaces. With this observation, we employ the technique by Eiseman et al. (Eisemann & Décoret 2006). Hardware voxelization is accomplished using two important observations:

1. a rendered viewport implicitly defines a grid as shown in Figure 2.12, and

2. whenever the GPU renders an object, it traverses every primitive and it also finds each cell intersected in this implicit grid.

At screen pixel (x, y), for every fragment generated, $(x, y, fragment\ depth\ or\ z)$ indicate a grid cell. Instead of discarding a fragment that is not visible, its information needs to be encoded in the red, green, blue and alpha (RGBA) color channels of the render target. Hence, for each grid cell intersected, its corresponding bit in the render target is set to 1. Using *logical OR* or *additive blending*, such 1-bit information about every fragment at each screen location is gathered. A texture with 32-bit precision can store up to 32 slices, and higher resolution can easily be obtained by using multiple render targets.

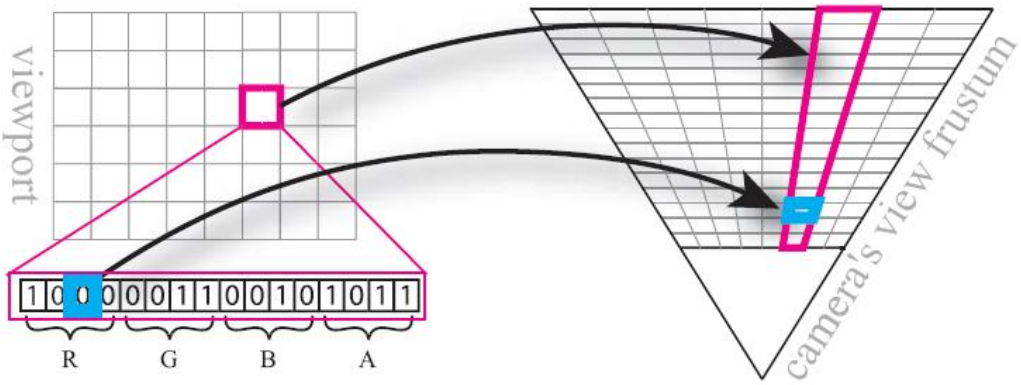


FIG. 2.12. Voxels grid encoding by Eiseman et al. (2006)

2.6 Other Related Work

In another approach, Purcell et al. (2003) implemented photon mapping on programmable graphics hardware to show the capability of commodity hardware to fully simulate global illumination. Instead of using an irregular structure like a *kd*-tree, they used a uniform grid structure which suitably maps onto GPUs. To accelerate searching of photons during the rendering pass, photons were kept sorted using a bitonic-sort or a stencil-routing method. Sorting on the GPU and photon searching involves many texture operations limiting the performance. To further improve interactive performance, Kruger et al. (2006) avoided any kind of intermediate radiance representation by using a screen-space photon tracing method that exploits the GPU's capability to render lines. They resolved intersections between objects and photon rays using layered depth maps and simple depth comparison in fragment shaders. Energy transfer was then done by rendering oriented sprites at each receiving point. Although this approach traces photons interactively, it requires multiple rendering of the scene to obtain layered depth images limiting its usability to simple scenes. Whenever a scene is rendered with no culling, the GPU processes all primitives that aren't culled by view frustum culling. Thus, all the necessary information about mul-

tiple interfaces is generated in a single rendering pass. However, it cannot be stored since Z-buffer or depth testing discards all the fragments that are not visible. Recently, Bavoil et al. (2007) produced interesting multi-fragment effects, such as translucency, by storing eight fragments per pixel in a K-buffer. Myers et al. (2007) extended this method to store more than eight fragments per pixel using a stencil-routed A-buffer stored as a multi-sample texture. These methods certainly avoid multiple renderings for complex scenes, however, they cannot deal with stencil-buffer overflow and also require an expensive sorting to be performed on all the fragments.

Chapter 3

APPROACH

3.1 Discussion of Goals

By reviewing previous work, we can conclude that the current refraction rendering techniques are limited to simulating refraction by pre-computing volumetric data on static objects (Ihrke *et al.* 2007) or to only two interfaces on dynamic objects (Oliveira & Brauwers 2007). The dynamic refraction technique by Sun et al. (2008) works at interactive frame rates, but only for simple scenes. This approach utilizes 3D textures and is limited by the amount of video memory on current graphics hardware. Low resolution volumetric data cannot deal with the subtleties on the surface of the object. All objects, refractive or non-refractive, must be enclosed inside the 3D texture, and light distribution also has significant storage requirements. Furthermore, it relies on CUDA for tracing photons through the volume. Our goal is to present an approach that interactively renders multiple refractions, thus caustics, using a traditional polygon-based rasterizing pipeline. For simplicity of implementation, we ignore phenomena such as scattering of light at each intersection. In this thesis, we present a method that simulates only the photon paths through a volume. Furthermore, we incorporate a computationally less-expensive technique, that is caustic mapping, with multiple surface refraction which enables us, unlike previous techniques, to render an object independently from its surroundings. This is a crucial requirement for

easy adoption of any technique into interactive applications such as games, as it provides the flexibility of programming every object in the scene with different rendering techniques.

Overall, our goals can be summarized as follows:

- Combine the best of image-space refraction and volumetric refraction techniques to achieve multiple refractions
  - Obtain a volumetric representation that is in the image-space
  - Use an interactive method to trace photons
- The method should fit into the existing rasterization-based pipeline
- Refractive objects must be rendered separately from the surrounding objects
- To be able to interactively visualize caustics, the method should work seamlessly with the existing caustics mapping techniques.

3.2 Overview of Method

Our method uses a volumetric representation of an object that contains only its occupancy information. We choose such a representation because it requires only one bit information per voxel grid. Unlike previous approaches using 3D texture, it enables us to obtain a large number of object slices in less memory. Current graphics hardware limits the size of 3D textures to $256 \times 256 \times 256 \approx 64MB$ of memory for a single floating point texture, requiring four bytes of memory per grid cell. Using just one bit per grid cell (one byte = eight bits), the same resolution can be obtained in just $2MB$ of memory. In addition, an entire scene at a higher resolution such as 1024×1024 or at a high definition resolution, such as 1920×1080, can be voxelized into 256 slices needing only $32MB$ and $\approx 64MB$ of memory respectively. The current limit on the dimension of 2D texture is 4096×4096.

Currently, GPUs support blending only for 32-bit color values which limits the number of slices to 256 ($32 \times 8MRTs$). However, with suitable hardware supporting 128-bit color blending, the number can increase up to 1024 ($128 \times 8MRTs$).

Furthermore, previous approaches depend upon interpolating the orientation of a surface by using a $3 \times 3 \times 3$ or $4 \times 4 \times 4$ neighborhood around the points where the light hits. Such techniques can generate a discrete set of surface normals. We, however, interpolate surface normals by using the depth images obtained from the volumetric data. Depth images allow for smoother interpolation of surface orientation. Additionally, the depth images that we obtain are already sorted in the viewing direction avoiding expensive sorting operations like those discussed in other works on GPU (Bavoil *et al.* 2007; Myers & Bavoil 2007). By using a high resolution for slicing, we ensure that the chance of overflow in the additive blending is greatly reduced. Features less than the size of one voxel would cause overflow and in such cases the location of the feature would be shifted by one voxel. Since we store depth images in different color channels of floating point textures, we can leverage the GPU's filtering capabilities to obtain even smoother depth values. Subsequently, by employing an image space ray tracing algorithm (Oliveira & Brauwers 2007), we find the intersections between these depth images and propagate light rays inside the object. We operate on 2D depth images of an object; hence, our algorithm inherits all the advantages of operating in the image-space. Additionally, as opposed to the approach by Sun et al. (2008) our method renders subtleties at least for front and back faces. Finally, using the existent caustics mapping techniques, we intersect light rays exiting from the back faces with the nearby diffuse objects to splat energy on them to produce caustics (Wyman 2008; Shah, Konttinen, & Pattanaik 2007).

3.3 Voxelization

We employ a surface voxelization algorithm (Eisemann & Décoret 2006) to obtain a volumetric representation of the object. To obtain a tightly bound voxel grid, we need the bounds of the object in each direction. Determining such bounds in real-time for a deforming mesh is computationally expensive; hence, we utilize per-pixel local bounds. As a first rendering pass, we rasterize the mesh without any culling or hidden surface removal and store minimum $(Zmin)$ and maximum $(Zmax)$ depth recorded for each pixel. This is accomplished in a single pass by writing the actual depth d and the reverse depth $(Zfar - d)$ to R and G channels of a texture and accumulating the result using GPU's *max* blend operation. In the next rendering pass, for each fragment generated, its corresponding local bounds $(Zmin)$ and $(Zmax)$ are then utilized to find its grid location. Although this technique generates a distorted grid, for most pixels it provides more compact slicing than uniform voxelization.

Similar to previous voxelization algorithms, we exploit the GPU's blending capability to obtain volume occupancy data in 256 slices using eight 32-bit textures as render targets. These 32-bit textures are then converted to two 128-bit unsigned integer textures. Using unsigned integer textures instead of unsigned normalized float avoids the extra computations needed each time during the conversion of normalized float into unsigned integer. Information about each occupied grid cell is encoded in these textures using GPU's blending capability. Since current Direct3D version does not support *logical OR* blending, we employ *additive blending*. Additive blending has a problem of overflowing when the two fragments intersect with the same cell; however, the occurrence of such scenarios with our algorithm is rare since we use a high resolution grid. Currently available GPUs do not support 128-bit unsigned integer blending. Thus, the voxelization resolution is limited to only 256 bits. However, with the development of more advanced hardware which supports

128-bit blending, we would be able to support voxelization of an object into 1024 slices to provide more accurate volumetric data and thus better rendering quality.

Despite its limitation, the GPU's blending capability enables us to obtain the required volume occupancy to peel up to 32 depth layers (Figure 3.2) and estimate surface normals at each occupied grid cell. We have observed that for voxelization at the desired output level, depth gradient of the nearby fragments is lost in the case of smoothly varying surfaces or surfaces with small curvature where multiple neighbors fall in the same slice. This is more applicable at high resolutions since the accuracy in depth cannot be increased accordingly because of the limit on the number of slices. Thus, with only 256 slices, it is desirable to voxelize any mesh into 256x256x256 or 512x512x256 grids.

3.4 Peeling Depth Layers

Current image space refraction algorithms are limited to refraction through only two interfaces. To obtain refraction through multiple interfaces, a depth-peeling technique can be utilized. Depth peeling can be described as peeling out layers of an object to uncover its internal details. It involves capturing multiple layers of fragments which are generated while rendering a geometry. Traditional depth peeling (Everitt 2001) captures one layer of fragment at each rendering pass by excluding previously captured layers. This is accomplished by recording per-pixel depth for each layer and then utilizing it in the next pass to reject fragments that are already drawn. However, this technique requires multiple renderings of the mesh which creates a bottleneck while dealing with complex objects. We have overcome this problem by peeling depth layers from the volumetric data obtained by voxelization (Figure 3.1). For ray tracing, as described in (Oliveira & Brauwers 2007), we need to compare the depth of a ray with a depth value from the image to determine whether the ray exits the object at that pixel. We identify the following two techniques to

accomplish depth layer peeling to obtain depth of a layer at each pixel. Figure 3.1 shows how layers are encoded in *texels* or texture pixels. For simplicity we show only 16 slices instead of 256.

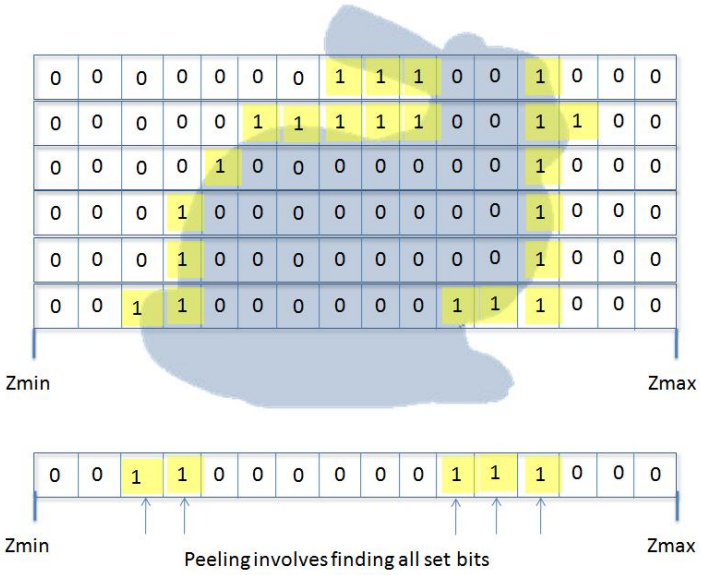


FIG. 3.1. Illustration of voxelization of the bunny model into 16 slices. Peeling depth layers simply involves finding the index of each set bit. A binary search can be performed to find each set bit

3.4.1 Peeling On-The-Fly

In this approach, we can begin ray tracing immediately after the voxelization step. To achieve this, we need to know the index of the depth layer that we are intersecting our light rays with. This problem is similar to finding the index b of the nth set bit in a 256-bit word as shown in figure 3.1. To accelerate this bitwise search we utilize lookup tables to store pre-computed answers for 8-bit values. Once we have index b of the layer n, we compute its depth as follows:

$$depth = Zmin + (b/255) * (Zmax - Zmin)$$

Where, *Zmin* and *Zmax* are either local bounds for a pixel or global bounds for the entire object.

We have observed that performing bitwise search on-the-fly for a pixel involves doing the same computations over and over again when multiple light rays are to be tested against it. This severely reduces the performance, as it contains several branching instructions and therefore becomes a hindrance in exploiting GPU's parallelism. We keep a separate pass to obtain depth layers as described below.

3.4.2 Separate Peeling Pass

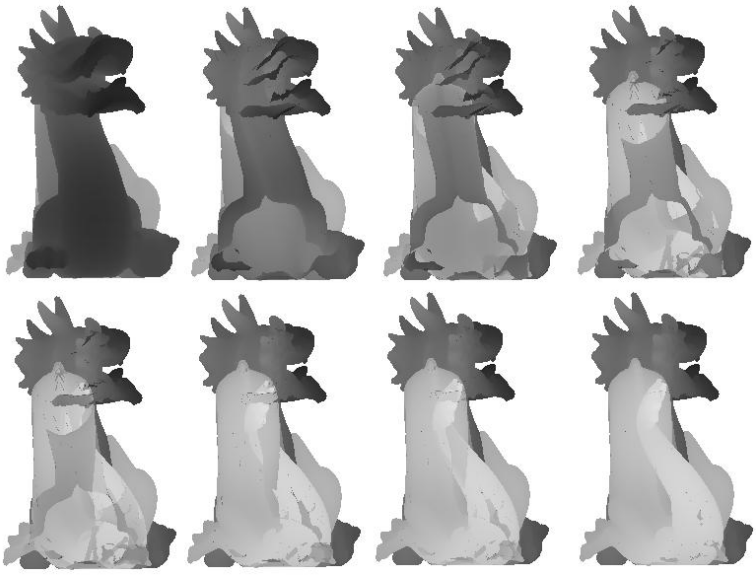


FIG. 3.2. First eight *layered depth images(LDI)* peeled for the dragon model (871,391 triangles). It can be easily observed that our method preserves subtleties on the surfaces.

By using a separate pass that involves only bitwise operations described above for each pixel and which stores the result in multiple render target textures would definitely avoid the problems associated with on-the-fly peeling. However, current graphics hardware allow only 8 simultaneous render targets, and in effect only 8 depth images. We overcome this problem by treating individual color channel of these render targets as separate depth images. This allows us to generate 32 depth images in a single pass. Another advantage of using this method, besides performance gain, is that hardware filtering can be easily applied while sampling these textures, thereby eliminating the need for filtering in a fragment shader and in turn preventing redundant texture accesses. To further accelerate this pass, we employ a binary search algorithm to find each set bit in 256-bit word. Such binary search would get more expensive than linear search when an object has more than 32 layers; however, we can ignore such rare cases since we can only peel up to a maximum of 32 layers. Thus, our choice of binary search is justified in almost all scenarios. Figure 3.2 shows first eight depth layers obtained for the dragon model with 871,391 primitives. Such an array of depth images is known as *layered depth images* or *LDI* array.

3.5 Surface Interpolation using Perspective-Corrected Depth Gradients

Whenever a light ray hits a surface, the path it will take is dependent upon the orientation or normal of the surface at the point of intersection. As discussed in the previous sections, we lose the information about the surfaces that are not visible in a particular view. We use depth images obtained in the previous pass to find surface normals at any required point. Several image-based rendering techniques frequently use the depth gradient method to calculate a normal as follows (Khan *et al.* 2006):

$$\nabla d(i,j) = (d(i+1,j) - d(i,j), d(i,j+1) - d(i,j))$$

$$\mathbf{g}_x = [1, 0, \nabla_x d]^T$$

$$\mathbf{g}_y = [0, 1, \nabla_y d]^T$$

$$\mathbf{n} = \mathbf{g}_x \times \mathbf{g}_y \tag{3.1}$$

where,

$d(i, j)$ represents depth at pixel (i, j) and **n** is the normal at that pixel. This is a crude approximation that leads to severe artifacts.

These artifacts are caused mainly due to two factors. First, the depth gradient is not scaled to the perspective of the view. Second, it is a forward difference technique which leads to non-smooth interpolation. We can avoid such artifacts by using a central difference method, but it does not solve the problem with the perspective view. This method assumes a unit change in both x and y directions between two adjacent pixels. However, in a perspective view this assumption is not valid. We eliminate this problem by calculating depth gradient around a pixel in view-space and then calculating normals for them as follows (Figure 3.3):

For each pixel (i, j) we have a reasonably accurate estimate of view-space depth $d(i, j)$ of the corresponding point from the eye. Let the perspective projection matrix used for rendering be M_p. Assuming some arbitrary perspective depth ϵ, we can project clip-space point $p^{'}(i, j)(2 * (i/width) - 1, 1 - 2 * (j/height), \epsilon)$ to view-space point $p(i, j)$ as:

$$p^{'}(i, j) = p^{'}(i, j) \times M_p^{-1}$$

$$p(i, j) = \frac{d(i, j)}{p^{'}(i, j)_z} * p^{'}(i, j)$$

This gives us the exact 3D coordinates of pixel (i, j). Similarly, we can calculate

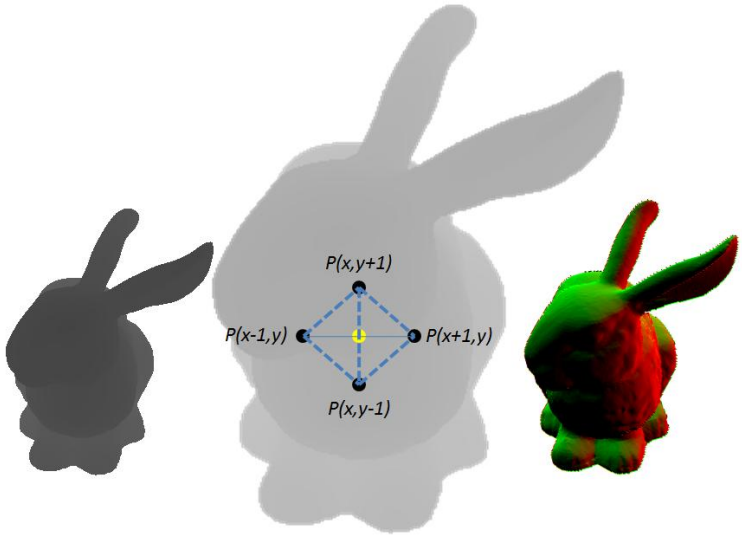


FIG. 3.3. To calculate surface normal at any point, gradients in vertical and horizontal directions are calculated. Since, the depth images provide reasonably accurate estimates of view-space depth; we can utilize them to find the actual 3D coordinates of each point shown and then calculate accurate bidirectional gradients.

$p(i+\delta, j)$,$p(i-\delta, j)$,$p(i, j+\delta)$ and $p(i, j-\delta)$. We can rewrite the equation 3.1 using bidirectional gradients and perspective correction as follows:

$$\nabla_x d(i,j) = \frac{d(i+\delta,j) - d(i-\delta,j)}{p(i+\delta,j)_x - p(i-\delta,j)_x}$$

$$\nabla_y d(i,j) = \frac{d(i,j+\delta) - d(i,j-\delta)}{p(i,j+\delta)_y - p(i,j-\delta)_y}$$

$$\mathbf{g}_x = [1, 0, \nabla_x d(i,j)]^T$$

$$\mathbf{g}_y = [0, 1, \nabla_y d(i,j)]^T$$

$$\mathbf{n} = \mathbf{g}_x \times \mathbf{g}_y \tag{3.2}$$

where, $\nabla_x d(i,j)$ and $\nabla_y d(i,j)$ are perspective corrected depth gradients in horizontal

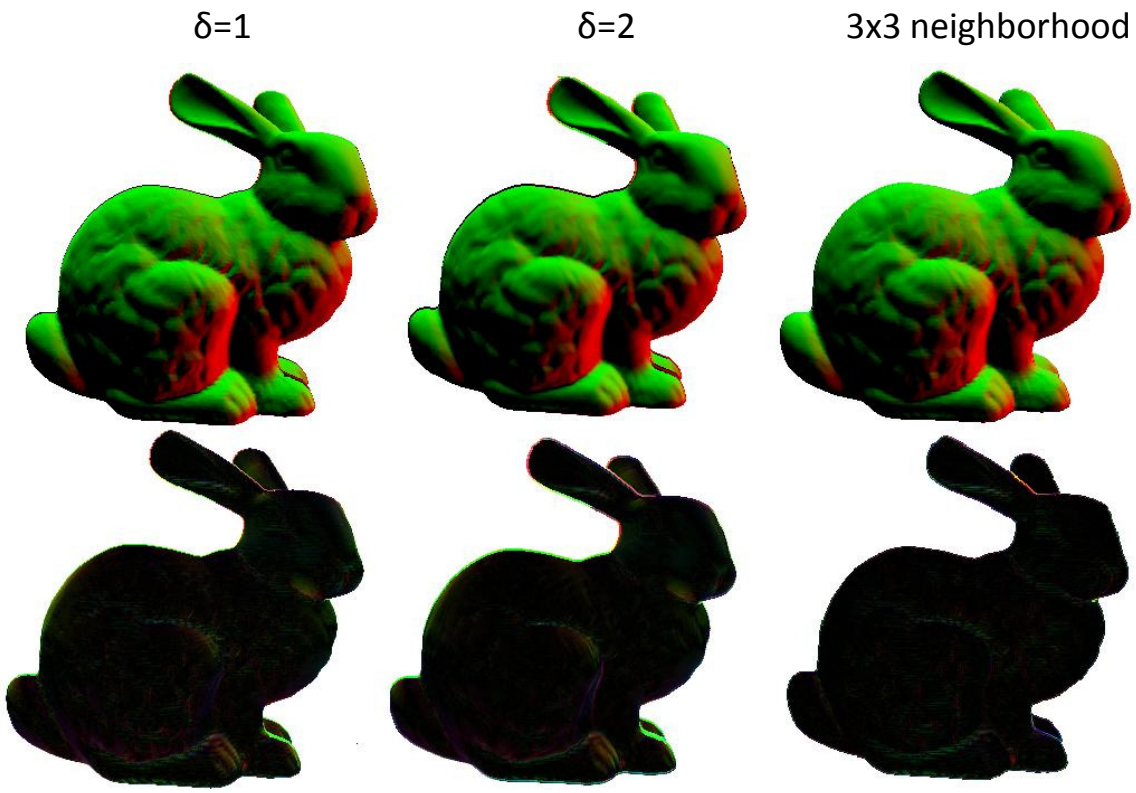


FIG. 3.4. Top row shows surface normals calculated using $\delta = 1$ (left), $\delta = 2$ (center) and mesh normals using mesh normals of a 3x3 neighborhood (right). Absolute deviation from the actual surface normals for these methods is visualized in the bottom row. For $\delta = 2$ errors are more prominent at the silhouettes of the object.

and vertical directions respectively, and δ is the integer step size in the horizontal or vertical direction.

Figure 3.4 shows the visualization of errors for the normals calculated using our method with $\delta = 1$, $\delta = 2$, and using the normals from a 3x3 mesh of vertices formed around each pixel. It can be observed that with $\delta = 2$, errors are smoother than with $\delta = 1$ and there is no considerable difference in the accuracy when compared to 3x3 neighborhood technique. Our method requires only 4 texture accesses, making it more efficient than 3x3 neighborhood which requires 9 texture accesses.

3.6 Image Space Ray Tracing

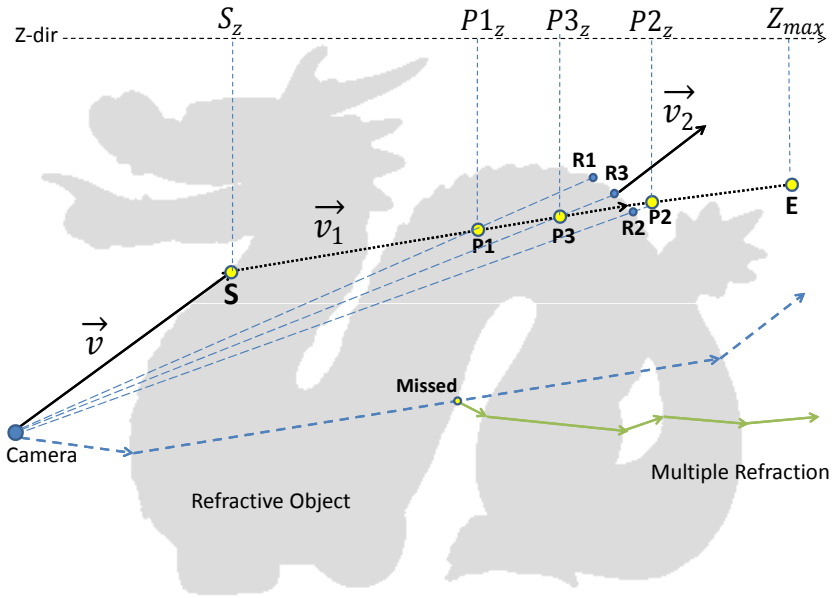


FIG. 3.5. Illustration of how binary search is performed with just two interfaces. A ray starting from the point S is traced to find an intersection with the backmost surface at point R3. All other faces that the ray $\vec{v_1}$ intersects are ignored. As an example, a ray with two intersections and multiple intersections is also shown.

We adopt the image-space binary search method proposed by Oliveira et al. (2007) to calculate intersection of the refracted rays with the depth maps. In any image-space approach, an object surface is usually represented by its depth map and its surface normals. Light rays generated are intersected against these depth maps. Figure 3.5 illustrates how binary search is performed against a depth map to estimate the intersection of refracted ray $\vec{v_1}$ with the back face. The search is restricted up to the point E with $z = Z_{max}$, where Z_{max} is the maximum depth of the object. The coordinates of E are given by

$$E = S + (\frac{Z_{max} - S_z}{(\vec{v_1})_z})\vec{v_1}$$

For any point $P(P_x, P_y, P_z, 1)$ in view-space along the ray $\vec{v_1}$, its perspective projection is given by $\tilde{P}(\tilde{P_x}, \tilde{P_y}, \tilde{P_z}, \tilde{P_w}) = M_p P$, where M_p is the camera's projection matrix. Furthermore, its corresponding texture coordinates t_P are given by

$$t_P = ((\frac{\tilde{P_x}}{\tilde{P_w}}) * 0.5 + 0.5, 0.5 - (\frac{\tilde{P_y}}{\tilde{P_w}}) * 0.5)$$

Using t_P, actual depth d_{t_P} of the surface can be fetched from the depth map. A simple comparison between P_z and d_{t_P} leads to following cases:

- $P_z = d_{t_P}$: Point P falls on the surface
- $P_z > d_{t_P}$: Point P is outside the object
- $P_z < d_{t_P}$: Point P is inside the object

To converge P to the point of intersection, a binary search is performed along the ray starting from S and ending at E to obtain points $P1$, $P2$, $P3$ and so on. A large number of iterations are required to reach the correct point of intersection. However, Oliveira et al. (2007) observed that in most of the cases, 5 such iterations provide a reasonably accurate

estimation of the intersection point. Figure 3.5 illustrates 3 such iterations to obtain points $P1$, $P2$, $P3$ and their corresponding intersection points $R1$, $R2$ and $R3$. Surface normal of the point $R3$ is then calculated using the texture coordinate t_{P3} of the point $P3$ and employing the perspective-corrected depth gradient method as described in the previous section. Refracted ray $\vec{v_2}$ is then generated by applying Snell's law.

To simulate refraction through multiple interfaces, the above discussed binary search can be applied at each interface recursively before a light ray leaves the refractive medium as depicted in Figure 3.6. For the first refracted ray $\vec{v_1}$ entered through point S, its corresponding intersection point with the first back facing interface ($R3$), can be found by performing the above discussed binary search on the first depth layer from the LDI array obtained in the peeling pass. Refracted ray $\vec{v_2}$ is then obtained using an interpolated normal at $R3$. In the next pass, this refracted ray $\vec{v_2}$ and point $R3$ become the incoming ray $\vec{v_1}$ and its starting point S. The search is again performed on the next depth layer from the LDI array. This process is repeated until $\vec{v_2}$ exits the object.

3.7 Restricting the binary search

At each interface, we utilize 5 iterations of binary search to find an intersection point. To converge to a correct intersection point in minimum number of iterations, each search must be restricted by a tighter $Zmax$ value. For each depth image in the LDI array, $Zmax$ represents the maximum depth for any pixel in it. Oliveira et al. (2007) obtain this value by performing a *parallel reduction* on a depth texture. A parallel reduction involves reducing a texture to half size by applying a reduction operation, in this case a $maximum$ or max operation, on 4 adjacent pixels. Figure 3.7 explains parallel reduction on a 4×4 texture. To perform parallel reduction on 32 high resolution depth textures becomes extremely expensive. Hence, we utilize *MIP levels* at the resolution of 256×256 of these textures for

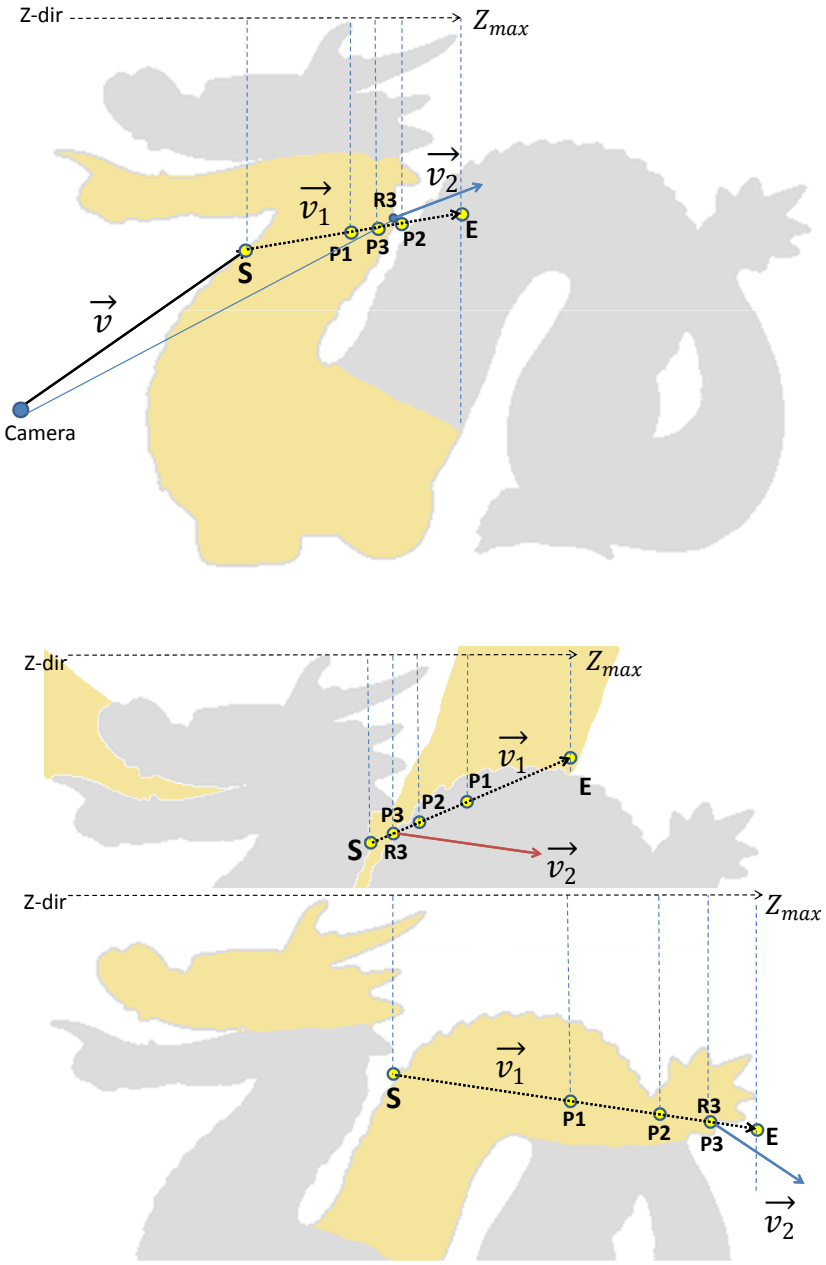


FIG. 3.6. In our method, binary search is applied at each interface of the object. This figure illustrates the binary search performed at the first back face of an object. Only first 3 iterations ($P1$,$P2$ and $P3$) are shown for simplicity. Region where the search is performed is also depicted. The search is restricted by the maximum depth, $Zmax$ of the current region. Once an intersection point $R3$ is obtained, a normal is calculated at that point and Snell's law is applied to generate the ray $\vec{v2}$

the reduction operation. A MIP is an image pyramid which contains versions of a texture that are at multiple lower resolutions. Using a low-resolution MIP level doesn't guarantee a correct maximum depth value of a texture; however, it provides a sufficient approximation. To reduce the error introduced due to this approximation, a small constant value can be added to the maximum depth obtained.

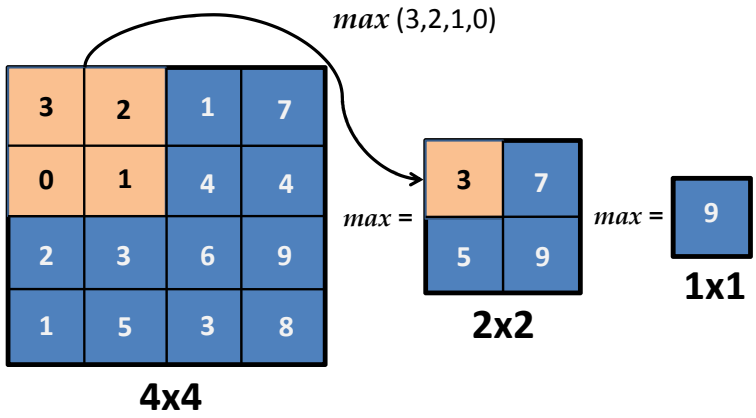


FIG. 3.7. Illustration of parallel reduction on a 4x4 texture.

It is also necessary to terminate the recursive binary search as soon as a photon hits the backmost face. We achieve this by utilizing a depth texture of the backmost interface that was obtained in the voxelization stage. If the depth of an intersection point approaches the corresponding depth on the backmost face within a threshold, the photon is marked as dead. Hence, for most photons, the search terminated after just a single iteration. Only the photons that travel through the region with a high depth-complexity are traced further.

3.8 Light Attenuation

Since our method can provide a good approximation for the distance travelled by a photon, light absorption can be easily simulated for a homogeneous object. Intensity I of an outgoing photon can be described by the Lambert's law of absorption as:

$$I = I_0 * e^{-\alpha x}$$

where,

I_0 is the intensity of the photon before entering the object,

α is the absorption coefficient of the material of the object, and

x is the optical path length of the photon.

Optical path length x for a homogeneous medium can be calculated as:

$$x = l * \eta$$

where,

l is the geometric distance travelled by the photon

η is the refractive index of the medium.

Light attenuation depends on the absorption as well as the scattering phenomena. Since, we assume the object to be homogeneous, scattering would occur only at the points where a photon hits the surface and not inside its volume. However, we do not store every point where a photon hits and thus cannot compute scattering due to photon hits. We use only absorption to approximate total attenuation.

Chapter 4

RESULTS

This chapter gives an in-depth analysis of our method, highlighting its important properties and also deals with the tangible limitations of this method. Our method simulates multiple refractions through a dynamic object in real-time. Here, we have analyzed the images created using our method. We also compare them with the images generated using only two-face refraction. We have also discussed the performance of our method depending on the complexity of the scene, voxelization resolution and different surface interpolation techniques.

4.1 Images and Discussion

All results discussed in this section are rendered with a viewport of 512×512, on a dual core AMD Athlon (2.6 GHz) processor equipped with NVIDIA 8800 GTX with 768MB of dedicated video memory. The system was implemented using Direct3D 10 APIs. All objects are dynamically voxelized into $512 \times 512 \times 256$ grid. As discussed in the previous section, binary search utilizes 5 iterations and is restricted by the bound computed by performing parallel reduction on 256×256 MIP levels. Furthermore, perspective-corrected central difference method is used for surface normal interpolation. Since we do not handle refraction of nearby geometry, refracted rays are looked up in an environment map. Refrac-

tion of nearby geometry can be approximated using the technique discussed in (2005b).

Our renderer achieves interactive frame rates ranging from 25 FPS to above 100 FPS depending upon the geometry and voxelization resolution. Unlike previous multiple refraction techniques, we can handle complex scenes with multiple refractive objects.

FIG. 4.1. Top row: Left image shows refraction just through the front and back interfaces of the dragon model and the right image shows refraction through multiple interfaces of the same model. Bottom Row: Left image shows refraction through first 2 interfaces of a sunflower model and the right image shows multiple refractions for the same. Refractive index of 1.25 was used for both the models.

Figure 4.1 shows the difference between multiple refraction and only two face refraction for the dragon model. A close observation of these pictures confirms that our method successfully traces light ray through multiple interfaces. For the sunflower model, we show

refraction through first two layers instead of front and back interfaces for the comparison. Interestingly, early photon termination can be easily observed at the regions with less *depth complexity*, that is the regions where light rays intersect with less number of interfaces before leaving the object.

FIG. 4.2. A vertex-displaced dynamic dragon model rendered using our method.

Figure 4.2 shows a deforming model rendered using our approach. To deform the model, we displace its vertices in a vertex shader. This displacement is performed for each rendering pass that needs to rasterize the model. For a model with a large number of vertices such vertex transformations may take several computing cycles. Direct3D 10 provides a way to save these transformed vertices in a separate memory buffer by using geometry

shader's *stream-out* functionality. However, since we focus on simply demonstrating the applicability of our approach to dynamic model, we do not implement such *stream-out* feature.

FIG. 4.3. Two Stanford bunny models rendered together in different views. This complex scene can be easily rendered by our approach at an interactive frame rate of 25 FPS. Top-left image with a single bunny is included for the comparison. Refractive index used is 1.2.

Our renderer can render multiple refractive objects that can fit into a particular view. We show that by rendering two Stanford bunny models together (Fig. 4.3). Since we voxelize our scene dynamically, such complex scenes can be rendered at interactive frame rates without any changes to our rendering pipeline. Multiple refractions through both the

bunny models are shown from different view angles.

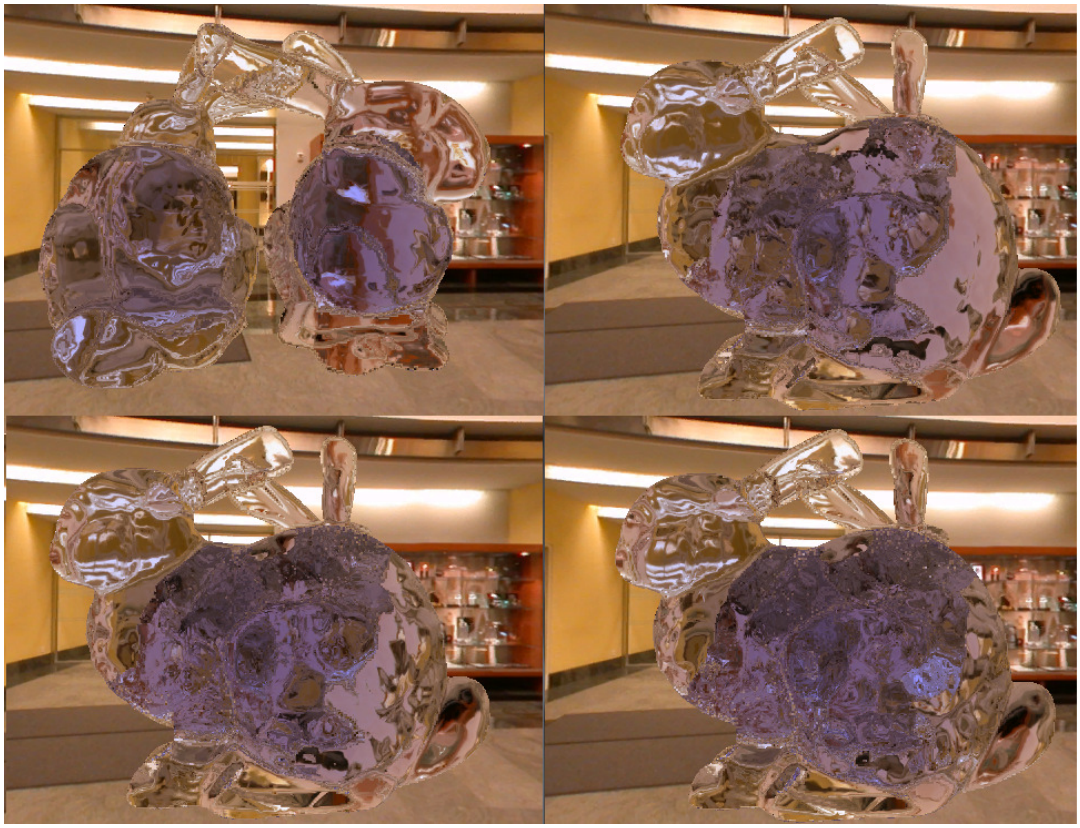


FIG. 4.4. Stanford bunny models rendered with an absorption coefficient of 4.0 (optical path length scaled to simulate actual geometric distance) for red and green color channels. Top row images are rendered with a refractive index of 1.11. Bottom row bunny models have a refractive index of 1.15 and 1.25 (left to right). Absorption coefficient is exaggerated just to visualize its effect.

Absorption can be seamlessly incorporated in our technique provided all the objects in the scene have the same optical properties like refractive index and absorption coefficient. All the occupied grids are assumed to be having uniform optical properties throughout. Figure 4.4 shows the final images rendered with a coefficient of absorption of 4.0 (not physical) and with different refractive indices. Since we use just a single bit per grid cell, empty or occupied, it is not possible to distinguish between two different objects in a scene.

Notice that, it will be possible to differentiate between the objects using multiple bits per grid cell, e.g. 2 bits can be used to differentiate between 3 objects by using 00 to indicate emptiness and 01,10,11 to indicate 3 different objects or 3 different materials. However, this would increase the memory needs of the algorithm. As a demonstration, we rendered the same scene of two bunny models by varying their absorption coefficient. For the images generated, absorption only for red and green colors is used.

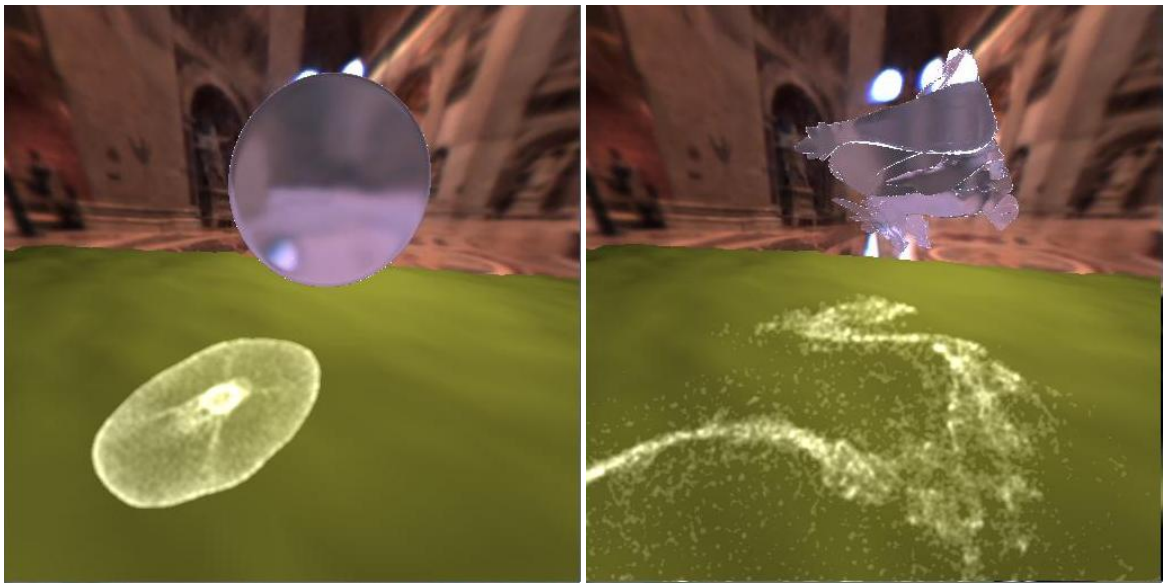


FIG. 4.5. Caustics produced by a sphere rendered to indicate the correctness of our photon tracing technique (left) and the same view is rendered using a refractive dragon (right).

Since our method operates independently of the surrounding objects, caustics mapping can be easily used along with it. However, caustics mapping has an inherent shortcoming that the refractive object needs to be rendered from the point of view of the light using a large view frustum. While rendering the actual scene if any point in the scene does not map to a corresponding point in the light-space, discontinuities in caustics can be easily seen, especially at the boundaries of the caustics map texture. We thus use a method by Kruger et al. (2006) which traces the photons in the eye-space. The method uses line

primitives that are indicative of the photons' starting positions and their directions. These primitives are generated in a geometry shader and the GPU rasterizes these line primitives extremely fast. All the intersections with the depth image of the surroundings are resolved in a fragment shader operating on each pixel of these lines. Kruger et al. used point sprites to splat energy, we, however, employ the image-space photon gathering method suggested by Wyman et al. (2006) for its simplicity. Figure 4.5 shows the caustics produced by a sphere on a bumpy floor. The purpose of this image is to indicate the correctness of our photon tracing method.

More results on caustics are shown in Fig. 4.6. For all caustics, photons were traced using our multiple surface refraction technique. Since multiple refractions would be expensive, we use single surface refraction for the final rendering pass and use the photons generated previously using multiple refractions to map onto the scene. We did not make any attempt to filter out noise from the caustics. Eliminating noise from the caustics is a well studied problem and for real-time methods, one can refer to (Wyman & Davis 2006). Some of the noise can also be attributed to the surface normal interpolation errors as described further in this chapter.

Surface normal interpolation using depth gradients is always associated with the errors in the interpolation of the normals at the silhouettes. Such errors become prominent when normals are interpolated on the peeled depth layers since the discontinuities increase in the regions where the photons are traced. Furthermore, errors due to discretization during voxelization also worsen these errors. This is illustrated in Fig. 4.7 which shows an enlarged view of the region near feet of the dragon model and also surface normals interpolated for one of the internal layers for the same region. Such inaccuracy can be avoided by utilizing an approach similar to the *variable size central difference operator* method (Shin 1999) or a *context-sensitive* surface interpolation (Shin & Shin 1995) which take local characteristics of the surface into consideration. Using voxelization at higher resolution may also solve

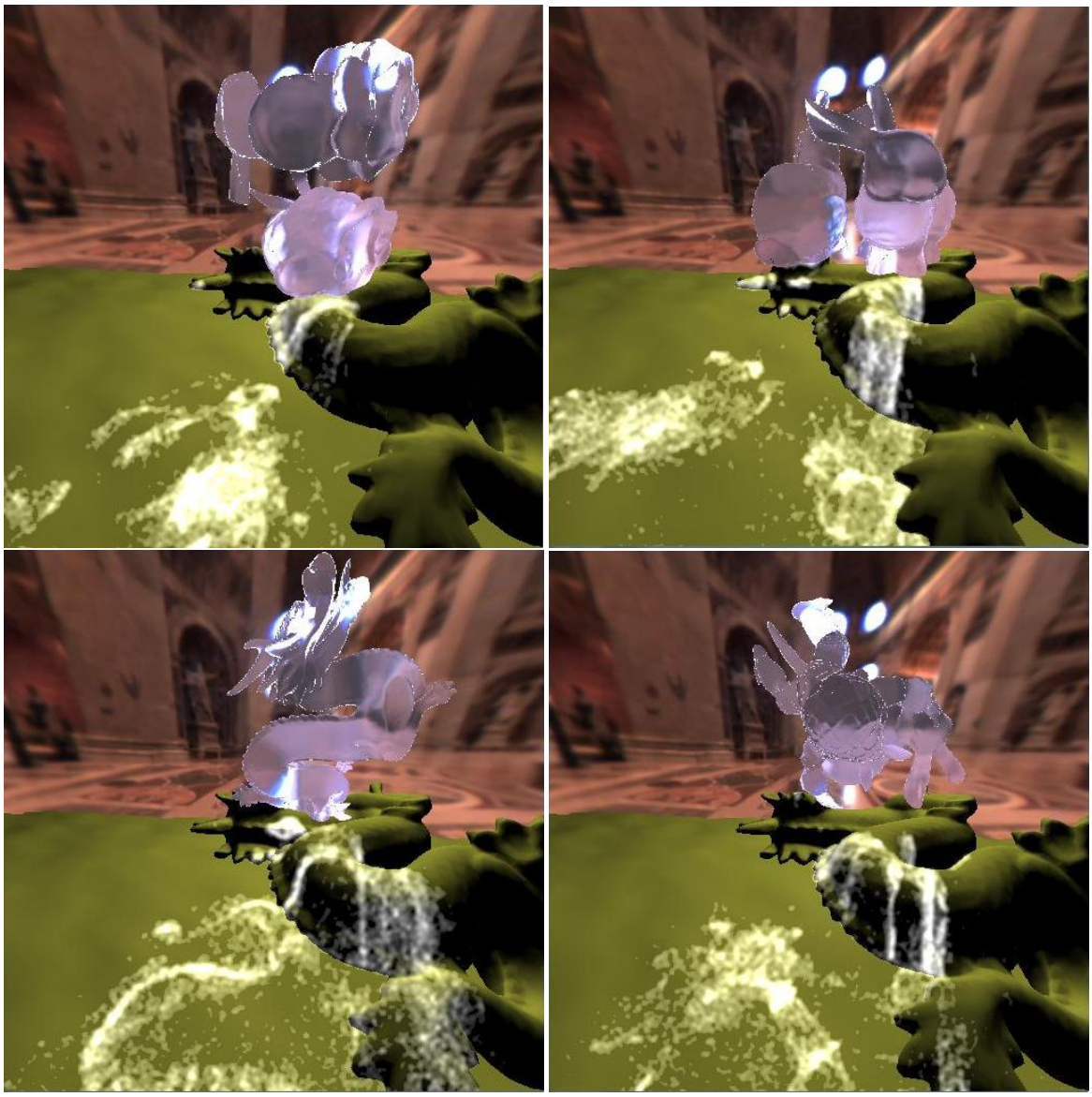


FIG. 4.6. Various complex scenes rendered at interactive frame rates using our method. As it can be seen, caustics can be easily rendered onto any surrounding objects. Top row: two bunny models rendered in two different views. Bottom row: a dragon model, an armadillo and a bunny casting caustics on a diffuse dragon model.

FIG. 4.7. Errors due to incorrect normals interpolation for minute silhouettes.

this problem, but it also requires increase in the number of slices. Current hardware limits 2D textures to 4096×4096 size; however, as the resolution increases, the number of slices should also be increased accordingly.

4.2 Performance

To measure performance of our system, we have used several standard meshes. However, we also generated corresponding low polygonal meshes to compare their performance against the original high polygonal versions. Performance statistics for two different voxelization resolutions are shown in Table 4.2; which indicate that we achieve targeted per-

MODEL	TRIANGLES	32 LAYERS		16 LAYERS	
		256X256	512X512	256X256	512X512
Holes	4,000	23	16	67	30
Dwarf	5,270	54	42	80	74
Bunny-lowpoly	15,000	106	57	114	71
Sunflower	18,094	54	43	88	80
Armadillo-lowpoly	30,000	59	37	95	58
Bunny and Bunny	30,000	48	25	90	41
Dragon-lowpoly	39,741	34	23	71	43
Armadillo and bunny	45,000	54	38	89	56
Ganesh	51,624	43	28	82	51
Buddha	67,234	53	28	83	52
Bunny I	69,451	87	36	99	60
Maxplanck	98,260	66	39	112	65
Bunny II	144,000	54	30	88	62
Igea	268,686	53	31	77	47
Armadillo	345,944	43	28	56	41
Dragon	871,391	31	21	43	29

Table 4.1. This table contains performance statistics for various models. Voxelization resolutions are also varied along with the number of depth layers utilized to trace photons through the model. All numbers indicate frames rendered per second on NVIDIA 8800 GTX with 768 MB of dedicated video memory.

formance. At both, 256x256 and 512x512 resolutions, the number of depth layers used for light propagation are also varied. All images are generated using a photon buffer of 512x512 size. We observed that the performance is mainly dependent on the depth complexity of a model and not the number of polygons in it. Since the depth complexity is view-dependent, we report the frame rates for the view comprising of the highest depth complexity possible for each scene.

For a 512x512 resolution texture, parallel reduction on 32 textures becomes a bottleneck. As we discussed previously, we leverage hardware's capability to generate a MIP level with the resolution of 256x256. We then perform parallel reduction on these 256x256

Stage	256x256x256				512x512x256			
	I-A	I-B	II-A	II-B	I-A	I-B	II-A	II-B
Min-Max Depths	2.085	0.365	2.086	0.369	2.101	0.428	2.112	0.404
Voxelization	2.235	0.652	2.176	0.622	3.807	1.358	3.076	1.312
Depth Layer Peeling	1.032	1.044	0.918	0.907	2.968	2.955	2.705	2.698
Parallel Reduction	0.573	0.58	0.574	0.564	1.294	1.266	1.323	1.306
Photon Generation	2.403	0.608	2.411	0.613	2.428	0.617	2.431	0.641
Photon Tracing	4.938	4.889	3.22	2.99	8.665	7.774	5.691	4.981

Table 4.2. Timings for each stage are measured at 256x256x256 and 512x512x256 voxelization resolution for low and high polygonal meshes in the scene containing two bunny meshes. Scene I-A: Two bunnies, 288,092 triangles. Scene I-B: Two bunnies, 30,000 triangles. Scene II-A: One bunny, 288,092 triangles. Scene II-B: One bunny, 30,000 triangles. All timings shown are in milliseconds.

textures to obtain depth bounds to restrict the binary search. We compared the performance of tightly restricted binary search using 5 iterations with a loosely restricted binary search using increased number of iterations for similar output. We could not see a performance gain by avoiding parallel reduction and increasing the number of iterations. Thus, the choice of performing parallel reduction is well justified. All frame rates and timings shown in this section are measured with the inclusion of parallel reduction stage.

Table 4.2 compares actual time taken by each pass of our algorithm to render two sets of a couple of scenes with 256x256x256 and 512x512x256 voxelization. It can be observed that, at a specific resolution, time taken by Depth Layer Peeling and Parallel Reduction remains fairly constant for a particular scene and it does not depend on the number of triangles in the scene. To compare the performance of parallel reduction at different resolutions, we have deliberately performed the reduction on 512x512 textures. Although not on the number of triangles, the time taken by Photon Tracing is dependent on the depth complexity of the scene. It can be observed that the time taken to trace photons for a single bunny scene is less than that for the scene with two bunny models. Photons are

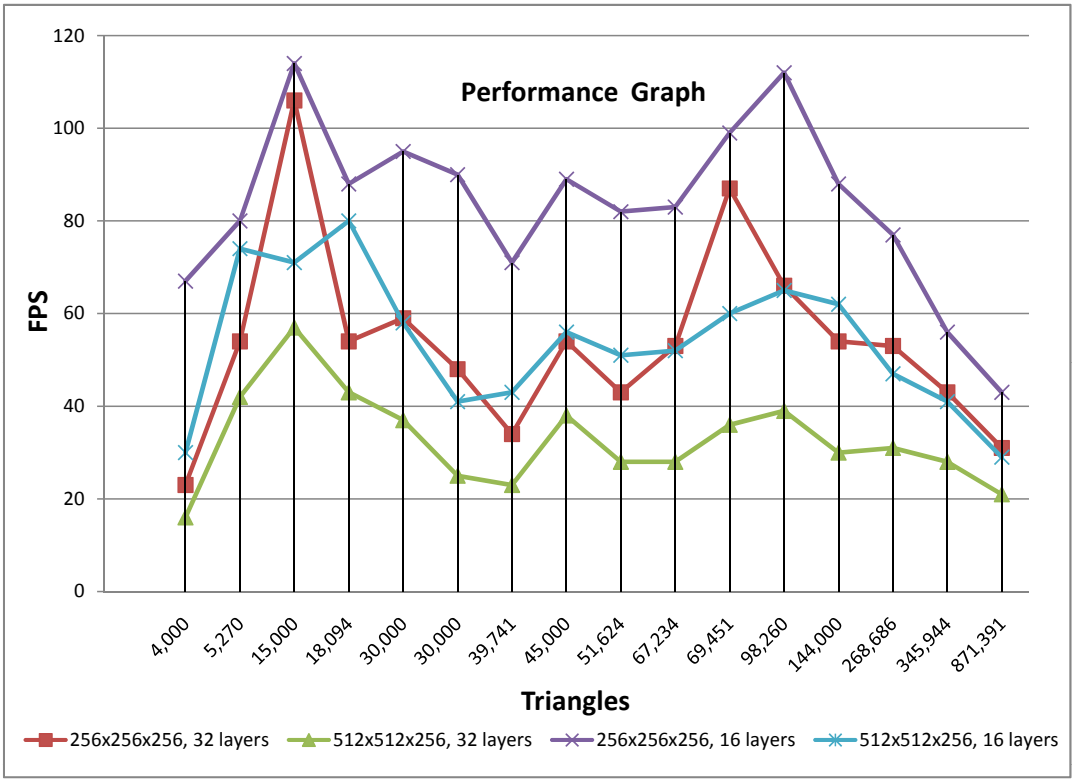


FIG. 4.8. The same performance data visualized as a graph. Non-linear relationship between the number of triangles and frame rate clearly indicates that the algorithm is not limited by the number of triangles in the scene.

traced through 32 depth layers by rendering a full screen *Quad* or a rectangle 32 times. At each rendering pass, we write all intersecting photons, dead or alive, to an output texture. Using a separate texture to write out dead photons would surely improve the performance of photon tracing stage.

The current implementation generates photons by rasterizing the geometry one more time. However, for a high polygonal model, this photon generation pass may become a bottleneck in the entire pipeline, referring to Table 4.2. More performance gain can be attained by using a *normal mapped* low polygonal proxy model. Normal mapping would

Factors	Wyman (2005)	Oliveira et al. (2007)	Ihrke et al. (2007)	Sun et al. (2008)	Our method
Refractions	Two faces	Two faces	Multiple	Multiple	Multiple
Surface details	Preserved	Preserved	Lost	Lost	Preserved
Interactivity	Very high	Very high	Low	Low	Medium
Storage requirement	Very low	Very low	High	Very high	Low
Separated from the surrounding	Yes	Yes	No	No	Yes
Multiple objects	No	No	No	No	Yes
Fits into raster-based pipeline	Yes	Yes	Yes	No	Yes
Pre-computations	Yes	No	Yes	No	No
Caustics	Mapped	Mapped	Inherent	Inherent	Mapped
Light attenuation	No	No	Yes	Yes	Yes

Table 4.3. Summary of the properties of our method and their comparison with the existing methods.

ensure that subtleties on the surface are preserved even for the surface represented by less number of polygons. Additionally, as mentioned earlier, we do not exploit the *stream-out* feature of the geometry shader stage which may save redundant processing of vertices and provide further performance gain.

Finally, the properties of our method are summarized and compared with the existing methods in Table 4.3.

4.3 Limitations

Although our method simulates refraction plausibly, there are two main limitations that should be mentioned. First, a more accurate surface normal interpolation method is required to generate more accurate and smooth normals in the regions of discontinuities and also at the silhouettes. As discussed previously an interpolation method adaptive to surface characteristics is required. Second, depth peeling pass, although accelerated by

a binary search on the bits, is still a hindrance while utilizing a voxelization resolution higher than 512x512x256. To perform bitwise search more efficiently, a better support for branching and bitwise operations is required on graphics hardware. With the fast paced advances in GPUs toward general purpose computing, such support may be available in the near future.

Currently, our implementation does not deal with the phenomenon of total internal reflection. Simulation of total internal reflection requires binding of all the depth textures to the rendering pipeline while advecting a photon. When a photon gets internally reflected, its path can be traced against the immediately preceding depth layer instead of the succeeding one. Since our depth images are stored as array of 2D textures in GPU memory, they need to be attached to corresponding texture samplers in order to access them. Although Shader Model 4.0 supports binding of 128 simultaneous texture samplers to a rendering pass, it does not support variable indexing into the an array of texture samplers. Thus, a depth layer cannot be accessed dynamically by simply varying its index. This can be accomplished by performing depth peeling in a geometry shader to store depth images into a 3D texture, where each slice in 3D texture would represent a depth image. It is also interesting to note that even for the refracted light rays which travel parallel to the screen, an image-space search doesn't provide correct intersections; since these rays intersect the triangles that are not facing the screen. A triangle at a silhouette is a good example. However, such triangles are viewed along their edges and do not contribute significantly to the final depth images. Thus, the light rays cannot be intersected with the pixels of such triangles and the algorithm consequently converges to false intersections.

Chapter 5

CONCLUSION

In this thesis, we have presented an algorithm to interactively simulate the phenomenon of refraction of light through multiple interfaces of deformable objects. We have also demonstrated that this algorithm can be easily applied to multiple refractive objects in a scene unlike previous approaches which have dealt with the scene with a single refractive object. Our method is essentially a three-pass method. In the first pass, we obtain volumetric data containing surface occupancy information by dynamically voxelizing objects which are represented as polygonal meshes. In the next pass, multiple layers or interfaces represented by depth images are decoded from this volumetric data. Our depth layer peeling technique allows for obtaining up to 32 depth layers which are already sorted from the viewing direction. Finally, photons generated from the eye are traced through all these layers using an adaptive binary search as the last pass. Our photon tracing is physically-based; however, we employ certain approximations to minimize the computational and storage resources that it needs. A few of them include: normal interpolation with a fixed gradient step, using a binary search over a linear search, and also the binary search is restricted by the bound calculated using parallel reduction over low resolution depth images. We have also discussed techniques to improve the performance at each rendering pass by exploiting the capabilities of current graphics hardware. Furthermore, unlike previous techniques,

our method is suitable for the standard raster-based pipeline and works with the caustics mapping techniques that allows us to render the refractive objects independently from the diffuse surrounding objects in a scene.

We expect our method to be more efficient in the near future when graphics hardware would provide better support for bitwise operations and branching instructions. Similarly, it will be possible to achieve total internal refraction with the availability of variable indexing of texture samplers which would allow for accessing any depth layer which is not in sequence. Also, 128-bit blending would allow for obtaining more slices providing more accurate depth images. We believe that our work opens up new avenues to explore various real-time methods to approximate surface characteristics by using variable size depth gradient operator or a context-sensitive method as mentioned in section 4.1. In addition to simulating refraction, we hope that our method to obtain sorted depth images proves useful for rendering other phenomena such as layered soft shadows.

REFERENCES

[1] Arvo, J. R. 1986. Backward Ray Tracing. In *ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, volume 12.

[2] Bavoil, L.; Callahan, S. P.; Lefohn, A.; Jo a. L. D. C.; and Silva, C. T. 2007. Multi-fragment effects on the gpu using the k-buffer. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 97–104. New York, NY, USA: ACM.

[3] Crane, K.; Llamas, I.; and Tariq, S. 2007. Real-time simulation and rendering of 3d fluids. In Nguyen, H., ed., *GPU Gems 3*. Addison Wesley. chapter 30, 633–674.

[4] Dong, Z.; Chen, W.; Bao, H.; Zhang, H.; and Peng, Q. 2004. Real-time voxelization for complex polygonal models. *pg* 00:43–50.

[5] Eisemann, E., and Décoret, X. 2006. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 71–78. ACM SIGGRAPH.

[6] Everitt, C. 2001. Interactive order-independent transparency. Technical report, NVIDIA Corporation.

[7] Génevaux, O.; Larue, F.; and Dischler, J.-M. 2006. Interactive refraction on complex static geometry using spherical harmonics. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 145–152. New York, NY, USA: ACM.

[8] Günther, J.; Wald, I.; and Slusallek, P. 2004. Realtime Caustics using Distributed Photon Mapping. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, 111–121.

[9] 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.

[10] Jensen, H. W. 1996. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, 21–30. London, UK: Springer-Verlag.

[11] Khan, E. A.; Reinhard, E.; Fleming, R. W.; and Bülthoff, H. H. 2006. Image-based material editing. *ACM Trans. Graph.* 25(3):654–663.

[12] Krüger, J.; Bürger, K.; and Westermann, R. 2006. Interactive screen-space accurate photon tracing on gpus. In *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering*, 319–330.

[13] Myers, K., and Bavoil, L. 2007. Stencil routed a-buffer. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, 21. New York, NY, USA: ACM.

[14] Oliveira, M. M., and Brauwers, M. 2007. Real-time refraction through deformable objects. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 89–96. New York, NY, USA: ACM.

[15] Purcell, T. J.; Donner, C.; Cammarano, M.; Jensen, H. W.; and Hanrahan, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 41–50. Eurographics Association.

[16] Shah, M. A.; Konttinen, J.; and Pattanaik, S. 2007. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics* 13(2):272–280.

[17] Shin, B. S., and Shin, Y. G. 1995. Fast normal estimation using surface characteristics. *Visualization '95. Proceedings., IEEE Conference on* 159–166, 449.

[18] Shin, B. S. 1999. Efficient normal estimation using variable-size operator. In *The Journal of Visualization and Computer Animation*, volume 10, 91–107. John Wiley and Sons, Ltd.

[19] Sun, X.; Zhou, K.; Stollnitz, E.; Shi, J.; and Guo, B. 2008. Interactive relighting of dynamic refractive objects. In *ACM Transactions on Graphics, to appear*.

[20] Wyman, C., and Davis, S. 2006. Interactive image-space techniques for approximating caustics. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 153–160. New York, NY, USA: ACM.

[21] Wyman, C. 2005a. An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics* 24(3):1050–1053.

[22] Wyman, C. 2005b. Interactive image-space refraction of nearby geometry. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 205–211. New York, NY, USA: ACM.

[23] Wyman, C. 2008. Hierarchical caustic maps. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 163–171. New York, NY, USA: ACM.