IMAGE-SPACE APPROACH TO REAL-TIME REALISTIC RENDERING

by

MUSAWIR SHAH

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering
at the University of Central Florida
Orlando, Florida

Fall Term
2007

Major Professor:
Sumanta Pattanaik

© 2007 by Musawir Shah

Abstract

One of the main goals of computer graphics is the fast synthesis of photorealistic image of virtual 3D scenes. The work presented in this thesis addresses this goal of speed and realism. In real-time realistic rendering, we encounter certain problems that are difficult to solve in the traditional 3-dimensional geometric space. We show that using an image-space approach can provide effective solutions to these problems. Unlike geometric space algorithms that operate on 3D primitives such as points, edges, and polygons, image-space algorithms operate on 2D snapshot images of the 3D geometric data. Operating in image-space effectively decouples the geometric complexity of the 3D data from the run-time of the rendering algorithm. Other important advantages of image-space algorithms include ease of implementation on modern graphics hardware, and fast computation of approximate solutions to certain lighting calculations. We have applied the image-space approach and developed algorithms for three prominent problems in real-time realistic rendering, namely, representing and lighting large 3D scenes in the context of grass rendering, rendering caustics, which is a complex indirect illumination effect, and subsurface scattering for rendering of translucent objects.

To my parents

TABLE OF CONTENTS

LIST OF FIGURES x

LIST OF TABLES xx

CHAPTER 1 INTRODUCTION 1

1.1 Real-time Performance and Interactivity 4

1.2 Image-space Approach 6

1.3 The GPU 10

CHAPTER 2 BACKGROUND 14

2.1 Radiometry and the Rendering Equation 14

2.1.1 Global Illumination 19

2.2 Light Source Representation 22

2.2.1 Directional Light Source 23

2.2.2 Point Light Source 24

2.2.3 Area Light Source 25

2.2.4 Environment Lighting 28

2.2.5 Light Maps 29

2.3 Appearance Modeling 30
2.3.1 Bidirectional Reflectance Distribution Function 32
2.3.2 Bidirectional Texture Function 38
2.3.3 Bidirectional Surface Scattering Reflectance Distribution Function 40
2.4 Reflection and Refraction 43
2.4.1 Caustics 45
2.4.2 Dispersion 46

CHAPTER 3 REAL-TIME RENDERING OF COMPLEX GEOMETRY 48
3.1 Bump Mapping and Displacement Mapping 49
3.2 Background 51
3.3 The Algorithm 53
3.4 Results and Analysis 56
3.5 Discussion 61

CHAPTER 4 LIGHTING IN LARGE, COMPLEX SCENES: REAL-TIME RENDERING OF REALISTIC LOOKING GRASS 66
4.1 Introduction 66
4.2 Related Work 69
4.3 Our Rendering System 71
4.3.1 BTF Acquisition 71
4.3.2 BTF Compression and Storage 73

4.3.3 BTF Rendering 75

4.4 Results 77

4.5 Conclusion and Future Work 78

CHAPTER 5 CAUSTICS MAPPING: AN IMAGE-SPACE TECHNIQUE FOR REAL-TIME CAUSTICS 80

5.1 Introduction 80

5.2 Previous Work 82

5.3 Rendering Caustics 85

5.3.1 Caustic formation 85

5.3.2 Caustics Mapping Algorithm 86

5.3.3 Creating the Caustic-Map 87

5.3.4 Convergence of Intersection Estimation Algorithm 91

5.3.5 Caustic Intensities 97

5.3.6 Implementation Issues 99

5.4 Results and Limitations 101

5.5 Conclusion and Future Work 104

CHAPTER 6 IMAGE-SPACE SUBSURFACE SCATTERING FOR INTERACTIVE RENDERING OF DEFORMABLE TRANSLUCENT OBJECTS 108

6.1 Introduction 108

6.2 Previous Work 110

6.3 Background . . . 113

6.3.1 Multiple Scattering . . . 114

6.4 Our Algorithm . . . 116

6.4.1 Algorithm Implementation Steps . . . 119

6.4.2 Environment Lighting . . . 122

6.4.3 Rendering Multiple Objects . . . 124

6.4.4 Comparison with Previous Work . . . 125

6.5 Results . . . 127

6.5.1 Analysis . . . 131

6.5.2 Reducing Temporal Artifacts . . . 132

6.6 Conclusion and Future Work . . . 134

CHAPTER 7 ARTIST DRIVEN INTERACTIVE EDITING OF HETEROGENEOUS TRANSLUCENT MATERIALS . . . 137

7.1 Abstract . . . 137

7.2 Introduction . . . 138

7.2.1 Editing translucent materials . . . 139

7.3 Analysis of the dipole diffusion model . . . 141

7.3.1 Changing σ_a and σ_s . . . 143

7.3.2 Changing g . . . 145

7.3.3 Changing η . . . 146

7.3.4 Perceived visual effects of scattering . . . 147

7.4 Artist driven model for heterogeneous translucent materials 149

7.4.1 Augmented dipole diffusion model for heterogeneous scattering 150

7.5 Results 152

7.6 Summary and future work 154

CHAPTER 8 SUMMARY AND FUTURE WORK 156

LIST OF REFERENCES 159

LIST OF FIGURES

1.1 Discrete region of the 3D scene visible through a pixel. 5

1.2 (a) Shadow ray: The point on the table is in shadow since its path to the light source is blocked by the sphere. (b) Shadow Mapping: Point p_s is in shadow since it is occluded in the light's view, whereas p_u is not in shadow since it is directly visible. 6

1.3 The rasterization pipeline. 10

2.1 Global Illumination. (a) First evaluation (direct illumination) of the rendering equation in which only the light source emits light. (b) Second evaluation (indirect illumination) of the rendering equation in which the reflected light from the first evaluation is considered as incident light. (c) Final image rendered using both direct and indirect illumination. Images courtesy of Juraj Obert. 19

2.2 Indirect illumination effects: (a) Color bleeding from the red and green walls onto the facing sides of the boxes. (b) Refractive caustics due to a glass container. Images courtesy of Jaroslav Krivanek. 20

2.3 Directional lighting. 22

2.4 (a) Point light, and (b) Spot light. The yellow region in the right image represents the illumination cone of the spot light. 24

2.5 Area light source. 25
2.6 Soft shadows due to area light source illumination. 26
2.7 Environment lighting representations: (a) Spherical Environment Map, and (b) Cube Environment Map. 27
2.8 Real-world environment maps captured at the Grace Cathedral: (a) spherical representation, and (b) cube map cross representation. Images taken from http://www.debevec.org. 28
2.9 Hierarchy of reflectance models. 30
2.10 The Bidirectional Reflectance Distribution Function. 31
2.11 Screen captures of our BRDF viewer application. 36
2.12 Discrete BRDF storage scheme on the GPU. (Left) Projection of a 3D vector onto a 2D plane. (Right) Selecting a pixel location using the view and illumination vectors. 37
2.13 Scattering in a back lit 3D object. 41
2.14 Incident light reflected off a (a) shiny surface, and (b) a perfect mirror. Notice that in the case of a perfect mirror, all the light is reflected in a single direction. (c) Light refracted by a perfect refractor. 43

3.1 A simple torus mesh (left) rendering with bump mapping (center) using a bump map texture (right). 49
3.2 A sphere (left) rendering with bump mapping (center), and displacement mapping (right). 50
3.3 3D height-field and view ray geometry used in Section 3.3 to illustrate the algorithm. 51

3.4 Linear search along the view ray. 53

3.5 Interval search along the view ray; for this specific example an intersection is found in two iterations. The number of iterations for any given case depends on the angle of the incoming view ray and the function it intersects with. 54

3.6 Interval mapping over an arbitrary polygon surface. 60

3.7 Empirical results" Images in the first column are rendered normally using interval mapping. Images in the second and third column show the number of pixel iterations required for convergence. The grey levels in these two columns are coded black for 1 iteration, up to white for a maximum of 10 iterations. The second column shows the results from interval mapping while the third column represents relief mapping. By comparing the average color value of the second column with the third, it is clear to see that interval mapping converges quicker. 64

3.8 Miscellaneous screenshots from our nature demo. The rocks at the bottom of the pond are represented by a single interval mapped polygon. 65

4.1 Grass rendering with silhouettes using our algorithm (left), and comparison of our method (center) to static texture mapping (right) 67

4.2 BTF dataset acquisition. The image plane is placed directly over a tiling of the grass mesh. The mesh is tiled to prevent discontinuities at the edges. Camera rays are shot through each pixel in parallel based on the current viewing angle being sampled. The intersection distances along the viewing rays are stored in a depth map which is used for silhouette and external occlusion determination. 68

4.3 Example of the grass mesostructure influencing the silhouette of the terrain along hill crests. 73

4.4 Individual grass blades occluding external objects in the scene. (Right) Detailed of the external occlusion effect. 76

4.5 (Left to right) Images rendered using 4, 40, and 140 coefficients for the reconstruction of the luminance channel of the BTF. 78

5.1 Image rendered using the caustics mapping algorithm. This result was obtained using double surface refraction (both for the appearance of the bunny as well as for the caustics) at the rate of 42 fps. 81

5.2 Photograph of caustics from a spherical glass paper weight using a desk lamp to emulate directional spotlighting. The caustics are formed on rough paper placed underneath the refractive object. 83

5.3 Diagram showing how multiple light rays can refract through an object and converge at the same point on a diffuse surface. 86

5.4 Diagram of the intersection estimation algorithm. The solid-lined arrows correspond to the position texture lookups. 89

5.5 Caustics on non-planar surfaces can be obtained using Caustics Mapping. The scene shows underwater caustics on a shark and a rugged seabed. 92

5.6 Caustics from a glass sphere (from left to right) using the distance imposters method, our caustics mapping, and photon mapping. 3 iterations of the respective intersection estimation algorithms were used for both the first and the second image. The distance imposter's method produced a frame rate of 160 fps whereas caustics mapping produced 245 fps on a GeForce 7800. 93

5.7 Difference images of (Left) the distance imposters method, and (Right) our caustics mapping method with photon mapping. The difference images were obtained from the results shown in Figure 6. 96

5.8 Frames from a water animation demo with caustics mapping. Random ”plops” are introduced in the water surface mesh at regular time intervals. Notice the caustics pattern and the corresponding shape of the water surface. 97

5.9 Caustics rendered using our algorithm for simple objects such us spheres (245fps) as well as complex ones such as the Stanford bunny (200fps) 99

5.10 Comparison of photon mapping (Left) and Caustics Mapping (Right) for a complex refractive object. 100

5.11 Effect of refractive grid resolution. The top image was rendered using a grid resolution of 64×64, whereas the bottom image was rendered using a grid resolution of 128×128. The gaps between point splats that are visible at grid resolution 64x64, are not noticeable at grid resolution 128x128 102

5.12 Reflective caustics from a metal ring. The desired shape that is observed in real life is obtained from the caustics-mapping algorithm. Notice the circular caustic band outside the left edge of the ring. 104

5.13 (Left) Colored caustics from a refractive bunny with light attenuation in different color channels. (Right) Spectral refraction effect obtained using the caustic mapping algorithm with different refractive indices for each of the three RGB color channels. 104

6.1 From left to right: bird model rendered (a) using Lambertian diffuse and Phong specular lighting, (b) our subsurface scattering algorithm (27 fps), and (c) an offline renderer (approx. 3 mins per frame). Note the translucency at the cheek and feet regions in the latter images. 109

6.2 (a) Dipole source approximation for multiple scattering. (b) Diagram of our rendering algorithm. The points x_p on the individual splats project on to the visible surface points x_o. The multiple scattering term is evaluated at each x_p by taking the distance between x_o and the corresponding x_i. The results are accumulated using additive alpha blending and stored at x_o. 113

6.3 Flow chart illustrating the scatter texture creation process. 119

6.4 (Top) Spherical harmonic coefficients are stored in an environment map after pre-rotating them into the local coordinate frame formed by the direction vector of the corresponding pixel in the environment map. (Middle) Depth peeling to capture all potential points scattering points, x_i. (Bottom) Since our algorithm operates in image-space, a splat centered at a point x_i on one object can scatter light to receiving points x_o on the other object. 122

6.5 From left to right, objects with increasing geometric complexity rendered with subsurface scattering using our algorithm at rates of 26fps, 26fps, and 24fps respectively. Notice the geometric complexity does not greatly influence the render time due to our image-space approach. 123

6.6 (Top) Graph showing the relationship between the mean free path length and the rendering frame-rate of the algorithm. As the mean free path length gets larger, the number of splats decreases and therefore the frame-rate increases. (Bottom) Graph showing the relationship between the mean free path length and the time taken to render individual splats. As the mean free path length gets larger, the splat size increases, and therefore takes more time to render. 129

6.7 Subsurface scattering in a back lit (left) and front lit (right) horse model. These images were rendered at 24fps at a resolution of 800×600 pixels. 129

6.8 (left) Shadows cast from external and self occlusion. (right) Close-up of the blurred shadow due to scattering. 130

6.9 Multiple objects with different scattering properties rendered at 25fps. Note that due to our image-space clipping technique, scattering from one object to the other is prevented even though the objects are adjacent. 131

6.10 From left to right: the bird model rendered using 32×32 samples at 31 fps, 16×16 samples at 65 fps, and 8×8 samples at 151 fps. Undersampling results in "patchy" artifacts because the splats do not overlap sufficiently. 132

6.11 The Stanford Bunny model made of a highly absorbing material. Notice how the back lighting effect at the silhouettes rapidly decreases toward the center of the model due to the high material density. This image was rendered at 22fps. 135

6.12 (Top) Multilayered material rendered using the multipole diffusion method with our algorithm. The material is made of a thin yellow layer and a green layer. The second and third images show the effect of increasing and decreasing the thickness of the yellow layer. (Middle) The Happy Buddha model rendered using our algorithm with subsurface scattering properties of jade. (Bottom) The bird model rendered using our algorithm under environment lighting: (left) Grace Cathedral and (right) St. Peters. 136

7.1 The effect of increasing the absorption coefficient, σ_a, on the scattering response of the dipole diffusion function. The other parameters used to create these graphs are: $\sigma_s = 2.0$, $g = 0.0$, $\eta = 1.2$. As the absorption coefficient increases, so does the total extinction coefficient, and the function falloff becomes sharper indicating that the effective scattering range is decreasing rapidly. 141

7.2 The effect of increasing the absorption coefficient, σ_a, on the scattering response of the dipole diffusion function, while adjusting the scattering coefficient to keep the extinction coefficient fixed at 2.5. Other parameters have also been fixed at same values: $g = 0.0$, $\eta = 1.2$. Note that increasing the absorption coefficient decreases the overall scattering response. 143

7.3 The effect of increasing the scattering coefficient, σ_s, on the scattering response of the dipole diffusion function, while adjusting the absorption coefficient to keep the extinction coefficient fixed at 2.5. Other parameters have also been fixed at same values: $g = 0.0$, $\eta = 1.2$. Note that increasing the scattering coefficient increases the overall scattering response. 144

7.4 The effect of changing the anisotropy term, g, on the scattering response of the dipole diffusion function. Other parameters have been fixed at the following values: $\sigma_s = 2.0$, $\sigma_a = 0.1$, $\eta = 1.2$. Note that as g increases, indicating increased forward scattering, the total scattering response decreases. 145

7.5 The effect of changing the refractive index, η, on the scattering response of the dipole diffusion function. Other parameters have been fixed at the following values: $\sigma_s = 2.0$, $\sigma_a = 0.1$, $g = 0$. Note that as η increases, indicating light propagation from coarser to denser medium, the total scattering response decreases. 146

7.6 The effect of decreasing the extinction coefficient (thus increasing the smoothness of the falloff of the scattering function) on the irradiance. As the extinction coefficient decreases, the effective scattering range increases. This results in the irradiance field getting more and more blurry. 148

7.7 Increasing the scattering coefficient, while fixing the extinction coefficient, makes the final rendered image brighter. Note the numbers each image is the increase in the scattering coefficient with respect to the corresponding image in Figure 7.6 . . 148

7.8 Increasing the blur in the final rendered image by scaling up and down the distance input to the dipole diffusion function has the same effect as modifying the extinction coefficient. 149

7.9 The artist starts with a non-translucent surface and starts painting translucency directly on the 3D mesh. Note the soft and blurred appearance of the painted regions. 150

7.10 Two different effects created with varying amount of blur. 151

7.11 Painting with a skin colored brush. This modifies the irradiance modulation texture, which controls the magnitude of the scattering response. The desired colored scattering effect is thus created by adjusting the scattering magnitude in each of the R, G, and B channels separately. 152

LIST OF TABLES

4.1 BTF reconstruction parameters 75

6.1 Symbols used in the multiple scattering computation using the dipole source BSS-RDF model. 112

6.2 Rendering performance statistics for translucent materials with varying mean free path lengths. 128

CHAPTER 1

INTRODUCTION

Over the last few years, the field of computer graphics has grown into a very active area of research and development. Since graphics caters to a large variety of vastly different fields, ranging from the entertainment industry to the medical arena, it constantly attracts research in different directions. For example, in the entertainment industry, the movie industry in particular, the emphasis is on realistic visual appearance. In simulation and training, motion capture and animation are important factors. Whereas in the medical field, visualization and imaging tools for volumetric data are vital components. Although all these different fields push towards their own respective subgoals in computer graphics, there are two main goals that are universal: realism and speed.

A great deal of effort and research has been invested in synthesizing photo-realistic images using computer graphics. In order for an image to appear "real" or "natural", the rendering process should mimic the transport of light and its interaction with different objects in nature. Different objects have different material properties that influence their interaction with light, and hence their visual appearance. For example, shiny objects reflect most of the light at the surface, whereas

transparent objects transmit light through them. In translucent objects, the light enters through the surface, gets scattered inside, and emerges again from the surface, whereas in highly opaque objects most of the light entering the object is immediately absorbed. Real world materials exhibit all these properties (absorption, reflection, transmission, and scattering) to some extent in different proportions. Therefore, in order to accurately simulate the interaction of light with these objects, their properties must be accounted for. Research in computer graphics and optical sciences has introduced a number of models for describing the appearance of different types of materials. Although these have been successfully employed in rendering visually convincing images, there are two main problems. First, certain materials, such as human skin, are more complex than others. The existing appearance models are not sufficient for all classes of materials. Therefore, accurate modeling of the interaction of light with these materials still remains a research challenge. Secondly, accurate simulation of complex light-material interaction generally translates to expensive mathematical computation. This in turn makes the rendering process slow.

Next to realism, speed is an important goal in computer graphics. In fact, in certain applications, such as computer games and simulation systems, speed is generally a more important criteria than the rendering quality. In interactive applications, the user expects a visual response as soon as he or she issues a command. Therefore the rendering system has to be fast enough to keep up with the user's requests. Due to this time constraint, the various aspects of the rendering process are considerably simplified in order to complete execution in the allocated time, which generally compromises the overall quality of the output. A lot of research has been focussed on developing acceleration techniques to speed up rendering by employing special data structures, perceptual

error metrics, and so on. However, the major boost was provided by the graphics hardware manufacturers with the introduction of the programmable GPU (Graphics Processing Unit), which spawned a whole new generation of rendering algorithms. Since the GPU can be programmed, a user can distribute the rendering workload between the CPU and the GPU. Furthermore, the GPU consists of multiple vector processors that operate in parallel, which, if utilized appropriately, can easily outperform the CPU by a wide margin in most cases. Development of real-time rendering algorithms that exploit the GPU has become a very active area of research in computer graphics. Researchers are producing results that are increasingly visually comparable to those produced using off-line renderers, but at orders of magnitude faster, thus neutralizing the compromise between rendering quality and speed.

The work presented in this thesis addresses the aforementioned goals of realism and speed in computer graphics. In particular, it tackles the problems of real-time realistic rendering of objects with complex surface structures and light interaction properties, and the optical phenomena they give rise to. Such objects and phenomena are mostly left out of interactive applications such as games due to their high rendering costs, but are highly desirable since they greatly enhance the overall visual appeal of the rendered images. In this work, we present efficient, scalable algorithms that produce visually convincing results at speeds practical for interactive applications. Deviating from the convention of operating on 3-dimensional geometry consisting of vertices, polygons, and edges, we explore the 2-dimensional image-space consisting of pixels to develop faster algorithms in this lower dimensional space. Our main contributions are in three different aspects of real-time realistic rendering, namely, representing and lighting large 3D scenes in the context of

grass rendering, rendering caustics, which is a complex indirect illumination effect, and subsurface scattering for rendering of translucent objects.

A brief discussion of the terms “real-time” and “image-space” is presented next in order to clarify and facilitate the reader’s understanding of the key concepts employed in this work.

1.1 Real-time Performance and Interactivity

The terms “real-time” and “interactive” are used quite frequently in computer graphics to describe algorithms and techniques. However, unfortunately these terms are defined rather loosely and sometimes are used interchangeably. Theoretically, a function or a process is deemed to be real-time if it finishes its execution and produces an output within a fixed amount of time, regardless of its input. In practice, however, whether a process is real-time or not is determined by the system measuring the response of the process. If the process completes execution at a rate faster than the system can measure the output, the system experiences no delay and hence observes real-time performance. In the case of computer graphics algorithms, for example, the measuring system is usually the display device or a visual observer. If an algorithm produces images (or frames) at a rate faster than or equal to the refresh rate of the display device, the display device perceives it as real-time.

In graphics literature, a particular fixed definition of real-time does not exist. Different authors use it to describe different levels of performances, though it is often observed that anything greater than 1 fps (frames per second) is considered real-time. However, graphics applications running at 1 fps are quite slow since the transition from one frame to another can easily be detected by human viewers. If a human user were to interact with such an application, he would experience long delays between his action and the displayed result. Applications that output frames at a faster rate such that the user does not experience any delay are categorized as "interactive", a subset of real-time applications. Generally, applications producing output at the rate of more than 20 fps are considered to be interactive, based on the average human response time.

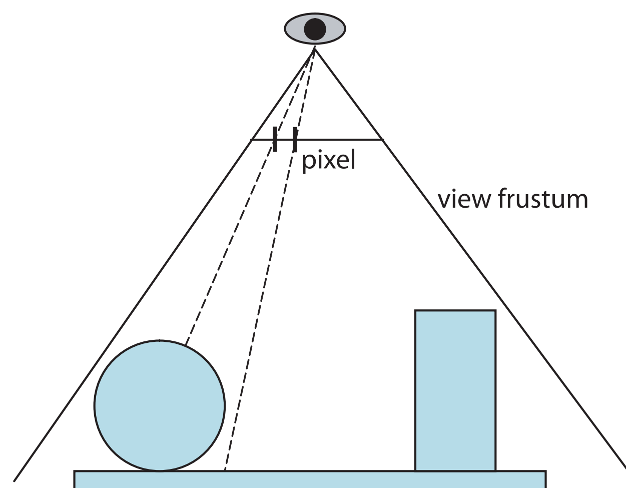


Figure 1.1: Discrete region of the 3D scene visible through a pixel.

1.2 Image-space Approach

An image-space approach is one that involves 2D images of objects rather than the 3D geometry. Generally, a 3D object or scene is represented in terms of geometric primitives such as vertices, polygons, edges, and surface patches. Geometric-space (also referred to as object-space) graphics algorithms directly operate on this data to produce a desired result. On the other hand, image-space algorithms first rasterize the 3D geometry into one or more 2D images, and then operate on the pixels of these images. The raserization step discretizes the 3D scene into pixels as visible from a given view. Each pixel in the image thus represents a finite region of the scene (Figure 1.1(a)). Therefore, all the information pertaining to that region required to perform the rendering can be stored at the corresponding pixel location in the image, such as 3D positions, surface normal vectors, etc.

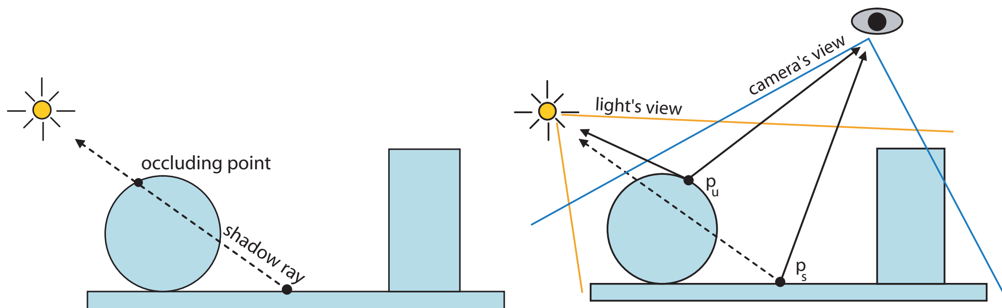


Figure 1.2: (a) Shadow ray: The point on the table is in shadow since its path to the light source is blocked by the sphere. (b) Shadow Mapping: Point p_s is in shadow since it is occluded in the light's view, whereas p_u is not in shadow since it is directly visible.

Consider the following example of rendering shadows under point light illumination to illustrate the difference between image-space and geometric-space approaches. In order to determine if a

particular point in the 3D scene is in shadow, one must check if there are any occluders blocking its direct path to the light source (Figure 1.2(a)). The geometric-space approach to solving this problem involves shooting a ray, known as the shadow ray, emerging at the point in direction of the light source and checking if it intersects any other object in the scene before arriving at the light source. The intersection checking process cycles through all the objects in the scene, utilizing their geometric information to determine if the ray intersects them or not. On the other hand, the image-space approach involves rasterizing the 3D scene from the light's view into an image called the "shadow map". Each pixel of the image stores the depth or distance of the corresponding region in the scene from the light source. The shadow map thus represents all the points in the scene that are visible from the light source (and hence receive light), and all the other points in the scene must be in shadow. Therefore, to determine if a particular point is in shadow, the point is first projected into the light's view and then checked to see if it exists in the shadow map. If it does, then it receives light, otherwise it is in shadow (Figure 1.2(b)).

Image-space algorithms have several advantages over geometric-space algorithms that make them suitable for real-time rendering.

Decoupled Run-time and Geometric Complexity: The most important feature of image-space algorithms is that their execution time is not dependent on the geometric complexity of the scene. Since they operate on images of the 3D scene, their run-time only depends on the size (or resolution) of the images. Whether the scene has one simple object or hundreds of complex objects does not influence the algorithm's run-time behavior. In context of the shadow example presented earlier, the shadow mapping algorithm involves querying a single shadow map, regardless of the

number and geometric complexity of the objects in the scene. Whereas in the geometric-space approach, the number of intersection tests performed for the shadow ray directly depends on the number of objects in the scene. Furthermore, the geometric complexity of the objects also determines if the intersection computation is simple and fast, or complex and slow.

Efficiency: Image-space algorithms are generally more efficient than geometric-space algorithms. This is because the input image data is first compiled from the geometric data during the rasterization step, and then the resulting pixels are processed by the algorithm. In doing so, only the information from the required parts of the 3D scene is extracted and the rest is discarded. Therefore, the number of computations performed are significantly reduced, which results in faster execution.

GPU Implementation and Storage: Image-space algorithms allow for very natural implementation on the GPU. The programmable GPU consists of a pixel processing stage in its pipeline that allows the user to perform operations on a per pixel basis. Since image-space algorithms involve operating on pixels of images, they are ideally suited for implementation on the GPU as opposed to geometric-space algorithms. In fact, the development of image-space algorithms for the most part is inspired by the GPU pipeline. Another advantage of image-space approach is that storage of processing data on the GPU is straightforward. The GPU offers storage in video memory in the form of 1D, 2D, and 3D arrays of four component vectors known as textures. Since image-space data generally consists of 2D images, it can easily be accommodated in the video memory. However, if a geometric-space algorithm, such as the shadow ray intersection technique, was to be implemented on the GPU, then the entire geometric information of the 3D scene must be somehow

packed into textures for storage on the GPU. Determining an efficient mapping from geometric to texture-space, and vice versa, itself is a difficult problem.

Of course, the image-space approach also has certain drawbacks.

Loss of Information: When a 3D scene is rasterized into a 2D image, some information is lost. Any regions of the scene that are not visible from the rasterization view are not accounted for in the image. Furthermore, the regions that are visible are represented as independent pixels in the image with no grouping or connectivity information. Therefore, for example, it would be difficult to identify the object a particular region belongs to. Notice that in geometric-space these problems do not exist since all the geometric information is available. They only arise when the geometric data is converted into image data. Fortunately, these issues can be overcome at the expense of storing extra image data. A 3D scene can be rasterized from multiple different views, thus resulting in multiple images, to ensure that all parts of the scene are accounted for. Similarly, extra data can be written at each pixel of the image, for example an object ID number, to identify the object it belongs to.

Artifacts: Image-space algorithms sometimes introduce visual artifacts in rendered images due to view space conversions. When a 3D scene is rasterized to an image, it is done so from a particular view. In the shadow example, the shadow map is rendered from the light's view. In most image-space algorithms, including shadow mapping, points are projected from one view space to another to query the image data. If the two views are fairly different, then the corresponding pixels will represent different sizes of the region of the scene, that is, a particular region in the image taken from one view might occupy several pixels whereas the same region in the image taken from the

second view might occupy only a single pixel. In such a case, when conversion from one space to another is performed, the pixels are "stretched", thus resulting in visual artifacts. In order to minimize these stretching artifacts, the image data should be rasterized from views close to the view of the final rendered image.

1.3 The GPU

The programmable Graphics Processing Unit or GPU is a critical component in real-time rendering. It is essentially a highly parallel vector processing system, streamlined for performing graphics operations. In order to understand the GPU, one must first understand the rasterization pipeline.

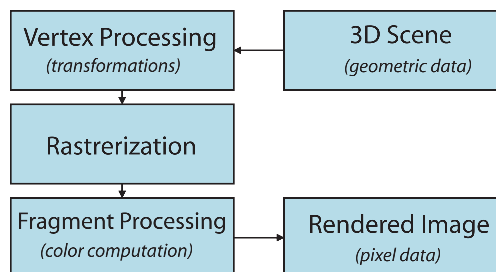


Figure 1.3: The rasterization pipeline.

Rasterization: Rasterization refers to the process whereby geometric data is converted into image pixels (Figure 1.3). A 3D scene consists of 3D objects, which are defined in terms of vertices and polygons (faces). Although graphics APIs allow users to specify arbitrary polygons, they are

eventually broken up into triangles for processing. The vertex and face data is passed on to the rasterization pipeline, where it undergoes a series of transformations. After this step, any geometry that is out of bounds of the camera's view is discarded, and the rest is projected onto the image plane. Each projected polygon thus occupies certain pixels of the image, which are then written to form the final rendering. If more than one polygon projects onto the same pixels, those pixels get written multiple times. In that case, the user can decide to either overwrite the previously written pixel value, or combine it with the new value according to some blending function.

The GPU implements this rasterization pipeline. Since operating on a vertex (or pixel) is independent of other vertices (or pixels), they are processed in parallel on the GPU. This makes GPUs much faster than CPUs where the processing is sequential due to hardware limitations. Programmable GPUs allow users to write code to control the vertex and pixel processing stages of the pipeline. Therefore, a user can dynamically control the transformations on a per vertex basis, and the values of the rasterized image on a per pixel basis. Each pixel of the image is represented by a four component vector. Generally, this vector is used to output a color value at each pixel in terms of four numbers corresponding to the red, green, blue, and alpha (transparency) channels so that the image can be displayed on a display device. However, since the pixel processing stage is programmable, a user can output any desired four component vector. For example, the user can output the 3D world position at each pixel of the corresponding point in the scene rather than its color. The reason why this is desirable is because the image data can be stored in the video memory of the graphics card instead of being displayed on the screen, and then later used to perform some task in

another render pass. Image-space algorithms consist of multiple render passes, in which first the different required data is collected in the initial render passes, and then used to synthesize the final image in the final render pass. In the case of shadow mapping, two render passes are required. In the first pass, the shadow map is constructed by rendering the 3D scene from the light's view. At each pixel of the shadow map image, the depth of the corresponding region in the scene to the light source is written. Then in the second render pass, the 3D scene is rendered from the camera's view. Each pixel in this rendered image is projected into the light's view space and the shadow map is queried. If the depth of the region of the camera image pixel from the light source is greater than the depth stored at the corresponding pixel in the shadow map, then the camera image pixel is in shadow.

Since GPUs are built according to the rasterization pipeline, they have certain limitations compared to the CPU, which prevents them from being an ideal platform for implementing arbitrary algorithms. Two of the main obstacles in GPU programming are storage and output control. The graphics card includes onboard memory, known as video memory, which is utilized by the GPU. Any data that is used on the GPU is stored in video memory in the form of 1D, 2D, and 3D textures, indexed by texture coordinates. This specialized storage makes it difficult to store arbitrary data. However, note that for image-space algorithms this is not a major issue since most of the data involved is in fact in the form of 2D images. The second obstacle in GPU programming is controlling the output. Write access to pixels is only obtained in the order that they are rasterized. Therefore it is not possible to write to an arbitrary pixel of an image, which is sometimes required, even in image-space algorithms. However, rasterization of the desired pixels can be "forced" by

rendering geometry at those locations in order to obtain write access. We employ this technique in our algorithms discussed later in the thesis.

CHAPTER 2

BACKGROUND

In this chapter, we present the theoretical background for realistic image synthesis. As mentioned earlier, in order to render realistic images, the interaction of light with objects in an environment as it occurs in nature must be accurately simulated. This simulation process involves three subproblems: representation of light, representation of light interaction properties of a particular material, and the actual interaction of light with the material. We will begin by describing the computations involved in simulating the interaction process, followed by a discussion of lighting and appearance models.

2.1 Radiometry and the Rendering Equation

The human visual system perceives objects in nature solely on the basis of how light interacts with them. The various properties associated with the appearance of an object, such as color

and shininess, can be attributed to the way light reflects off of the surface of the object towards the observer. When light arrives at the surface of an object, it undergoes a number of different interactions both at the surface and inside the object. Ultimately, it exists at the surface, perhaps greatly attenuated. This exiting or reflected light is what an observer's visual system utilizes to perceive the appearance of the object. In the same spirit, realistic rendering of a 3D scene requires solving for the exiting light, given the reflection properties of the objects in the scene, as well as the illumination conditions.

Before we look at the actual mathematical equations for computing the exiting light, we will describe some formal terminology and physical attributes of the various components involved in illumination and rendering.

Up until now, we have simply referred to all forms of illumination as "light". Physically, light is a form of energy known as "radiant energy" and is measured in Joules. Perhaps the more important unit for our discussion is the "radiant flux", which is defined as the flow of radiant energy through a surface per unit time. The radiant flux is denoted by the symbol Φ, and is measured in Joules/second or Watts. All the other terms used in rendering for describing light are written in terms of the radiant flux:

Irradiance: radiant flux per unit surface area (Watt/m^2)

$$E = B = \frac{d\Phi}{dA} \tag{2.1}$$

where E and B are the irradiance and the radiant exitance, respectively.

Intensity: radiant flux per solid angle (Watt/sr)

$$I = \frac{d\Phi}{d\omega} \tag{2.2}$$

Radiance: radiant flux per solid angle per unit projected area (Watt/m^2sr)

$$L = \frac{d^2\Phi}{d\omega dA_\perp} = \frac{d^2\Phi}{d\omega dA cos\theta} \tag{2.3}$$

The solid angle, $d\omega$, is the 3D counterpart of the 2D angle, $d\theta$, and is defined as:

$$d\omega = \frac{dA_\perp}{r^2} = \frac{dA cos\theta}{r^2} \tag{2.4}$$

where A is the area of the surface subtending the solid angle at distance r. Note the similarity with the 2D angle, which is defined as:

$$d\theta = \frac{ds}{r} \tag{2.5}$$

where s is the arc length. The solid angle at a given point essentially measures how big a surface area appears to an observer at that point. It is measured in *steradians* (sr).

The different terms representing the forms of illumination that we have just described are employed in different scenarios in rendering. Therefore, sometimes the illumination information has to be

converted from one form to another so that it can be applied appropriately in the given situation. These conversions can be derived directly from their definitions. For example, the irradiance, E, at a point with a differential area, dA, due to a light source emitting energy with intensity, I, is computed as follows. We begin with the definition of intensity:

$$I = \frac{d\Phi}{d\omega} \tag{2.6}$$

$$I = \frac{d\Phi}{dA cos\theta / r^2} = \frac{d\Phi}{dA} \frac{r^2}{cos\theta} \tag{2.7}$$

$$I = E \frac{r^2}{cos\theta} \tag{2.8}$$

$$E = \frac{I cos\theta}{r^2} \tag{2.9}$$

Rendering an image of a 3D scene involves computing the outgoing radiance at all the visible points of the scene, in the direction of the observer. The amount of outgoing radiance depends on the reflectance properties of the surface being rendered, and is defined by the reflectance function, f_r:

$$f_r(\omega_i, \omega_o) = \frac{dL(\omega_o)}{dE(\omega_i)} = \frac{dL(\omega_o)}{L(\omega_i) cos\theta d\omega_i} \tag{2.10}$$

We will now describe the “rendering equation”, which computes the outgoing radiance from the incoming radiance. The rendering equation is defined as:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L(x, \omega_i) f_r(x, \omega_i, \omega_o)(\omega_i \cdot n_x) d\omega_i \tag{2.11}$$

The outgoing radiance is computed for a point, x, and in the direction, ω_o. It is the sum of the emitted radiance, L_e, and the total reflected radiance given by the integral over the hemisphere, Ω, of incoming directions, ω_i. The reflectance function, f_r, gives the amount of reflected radiance in direction ω_o due to radiance incident at point x from direction ω_i. f_r is chosen based on the reflectance properties of the material being rendered. A detailed discussion of reflectance functions is presented later in this chapter in the context of appearance modeling. Finally, n_x is the normal vector at point x. The rendering equation also consists of a boolean “visibility function”, which indicates whether the point x receives light from a particular direction ω_i, or if its occluded by another object in the scene. This function, therefore, is responsible for producing shadows. The visibility function is defined in terms of the light source, and we introduce it into the rendering equation during our discussion of light source representation in Section 2.2

Figure 2.1: Global Illumination. (a) First evaluation (direct illumination) of the rendering equation in which only the light source emits light. (b) Second evaluation (indirect illumination) of the rendering equation in which the reflected light from the first evaluation is considered as incident light. (c) Final image rendered using both direct and indirect illumination. Images courtesy of Juraj Obert.

2.1.1 Global Illumination

The rendering equation essentially models the interaction of light with a 3D scene by relating the incoming radiance to the outgoing radiance. The reflected radiance is computed by considering incoming radiance from all directions of the hemisphere at a particular point. It is important to note that, according to the rendering equation, the reflected radiance from one point in the scene can contribute to the incoming radiance at another point. Consider a 3D scene with several non-emitting objects and one emitting light source, as depicted in Figure 2.1. When the rendering equation is first evaluated at the visible points of the scene, the only incoming radiance is from the light source. Now, if the rendering equation is evaluated a second time on the resulting illuminated scene from the first evaluation, the reflected radiance from the first evaluation can contribute to the incoming radiance of the current evaluation. In other words, in the first evaluation of the rendering equation, the 3D scene was illuminated *directly* by the light source. In the second evaluation, the

scene was illuminated *indirectly* by the light emerging from the light source and first reflecting or bouncing off different regions of the scene. The direct and indirect illumination is collectively referred to as global illumination. Subsequent evaluations of the rendering equation add to the indirect illumination. Since the reflected radiance reduces with each bounce, the contribution to the indirect illumination becomes negligible after a certain number of evaluations of the rendering equation. In practice for rendering purposes, two to three bounces are considered sufficient.

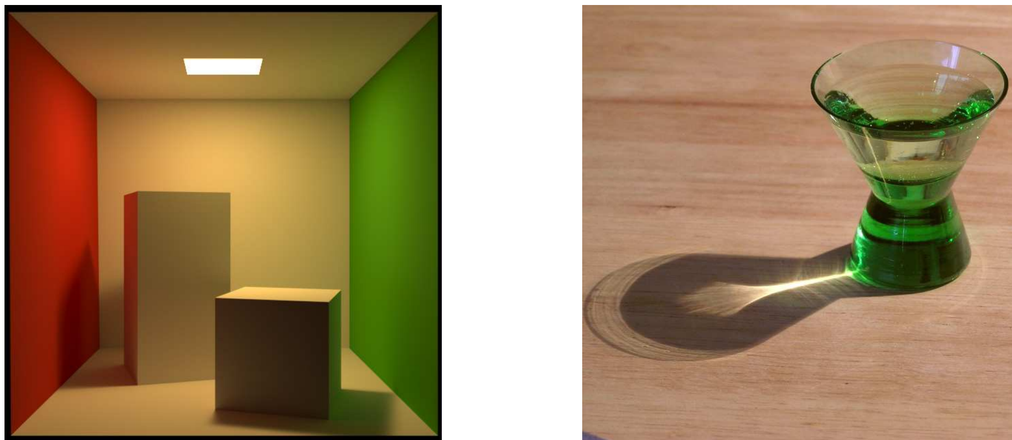


Figure 2.2: Indirect illumination effects: (a) Color bleeding from the red and green walls onto the facing sides of the boxes. (b) Refractive caustics due to a glass container. Images courtesy of Jaroslav Krivanek.

Accounting for indirect illumination is crucial in rendering photo-realistic images. It produces a number of important visual effects, such as *color bleeding* (Figure 2.2(a)) and *caustics* (Figure 2.2(b)), that greatly enhance the overall visual appeal of the image. These effects cannot be obtained with just direct illumination. However, evaluating the rendering equation for indirect illumination is an expensive process. This is because unlike the case of direct illumination in which the incoming radiance is received only from the light source, every point in the scene can potentially contribute to the incoming radiance at every other point. Due to this, indirect illumination is

generally not feasible for real-time applications and therefore the associated visual effects are left out.

Real-time indirect illumination is currently a very active topic of research in computer graphics. The most popular line of algorithms proposed in this direction are the pre-computation based solutions. These generally involve reshaping the rendering equation such that most of the expensive computations can be pre-computed and stored via an off-line processing step. Then, during the actual rendering stage, the pre-computed data is read and the final result is computed. Although these techniques account for indirect illumination as well as produce real-time frame rates, they impose a number of restrictions on things such as the viewing direction and the movement of objects in the scene. This is due to the fact that the pre-computation is performed for a particular configuration of the scene, including lighting and viewing, and it becomes invalid as soon as that configuration changes. However, depending on the algorithm, pre-computation based techniques are able to provide certain degrees of flexibility, for example change in viewing and illumination, but they are generally not sufficient for fully interactive applications such as games.

In addition to the pre-computation based solutions, there exists a line of algorithms that focus on isolating specific indirect illumination effects and computing them independently. These effects are then added in with the direct illumination to make the final rendering appear more realistic. This approach has two advantages. First, no global restrictions are imposed on the overall rendering process. Second, certain effects can be solved *approximately* to speed up the computation, without compromising the accuracy of other illumination effects. We demonstrate this strategy of

independently computing indirect illumination effects in our real-time caustics rendering algorithm described in Chapter 5.

2.2 Light Source Representation

We now describe how light sources are represented in computer graphics, and their influence on the rendering equation for computing the final outgoing radiance. Note that light sources contribute only to direct illumination, and therefore indirect illumination is ignored in this discussion.

The representation of light sources can be divided up into two main categories: *geometric representation* and *image-based representation*. We will first discuss the popular geometric representations of light sources.

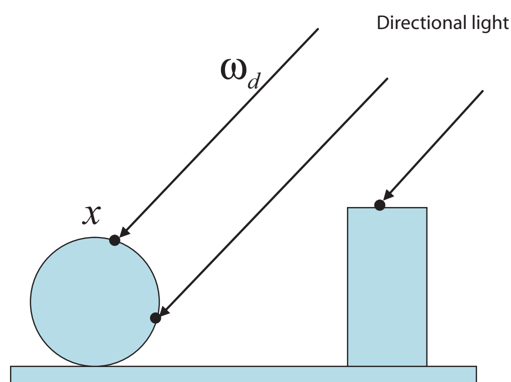


Figure 2.3: Directional lighting.

2.2.1 Directional Light Source

A directional light source, as the name implies, illuminates an entire 3D scene from one specific direction, ω_d (Figure 2.3). It is generally used to represent lighting from the source at large distances from the scene. For example, sunlight is usually represented as directional light. Employing a directional light source greatly simplifies the rendering equation:

$$L(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega B(x,\omega_i)\delta(\omega_i - \omega_d) f_r(x,\omega_i,\omega_o)(\omega_i \cdot n_x) V(x,\omega_i) d\omega_i \tag{2.12}$$

$$L(x,\omega_o) = L_e(x,\omega_o) + E(x,\omega_i) f_r(x,\omega_i,\omega_o)(\omega_i \cdot n_x) V(x,\omega_i) \tag{2.13}$$

where $\delta(\omega)$ is the Dirac delta function defined as:

$$\delta(\omega) = \begin{cases} \text{inf} & \text{if } \omega = 0, \\ 0 & \text{if } \omega \neq 0 \end{cases} \tag{2.14}$$

and ω_d is the source light direction. Notice that we have introduced the visibility function, V, in the rendering equation. It is a boolean function that indicates whether or not the point x is occluded in direction ω_i by other points in the scene. If it is, then V returns value 0, otherwise it returns 1.

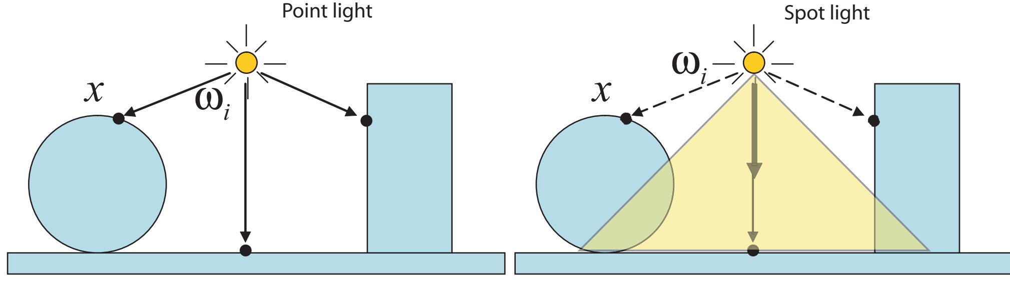


Figure 2.4: (a) Point light, and (b) Spot light. The yellow region in the right image represents the illumination cone of the spot light.

2.2.2 Point Light Source

A point light source is represented with a single 3D point in the scene, generally placed in close proximity of the other objects of the scene unlike the directional light source (Figure 2.4(a)). The rendering equation also simplifies under point light illumination since only a single illumination direction is used at each point, x. However, the direction changes with the position of the point.

$$L(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega \frac{I(\widehat{l-x})}{||l-x||^2}\delta(\omega_i - (\widehat{l-x}))f_r(x,\omega_i,\omega_o)(\omega_i \cdot n_x)V(x,\omega_i)d\omega_i \tag{2.15}$$

$$L(x,\omega_o) = L_e(x,\omega_o) + \frac{I(\widehat{l-x})}{||l-x||^2}f_r(x,\widehat{l-x},\omega_o)(\widehat{l-x}\cdot n_x)V(x,\widehat{l-x}) \tag{2.16}$$

$$\tag{2.17}$$

where l is the 3D position of the point light source, and $\widehat{l-x}$ is a unit vector indicating the illumination direction. In other words, the direction vector, ω_i, is computed dynamically and is given by $\widehat{l-x}$. Point light sources are often used in two different variations: omni-directional, and spot

light. In the case of omni directional lighting, the point light source illuminates in all directions. However, in the case of spot light, the point light source has a principal "view" direction associated with it. This direction forms a cone of possible illumination directions around it that specifies the illumination cut off (Figure 2.4(b)). If a point in the scene forms a direction vector with the light source that falls outside of this cone, the point receives no light.

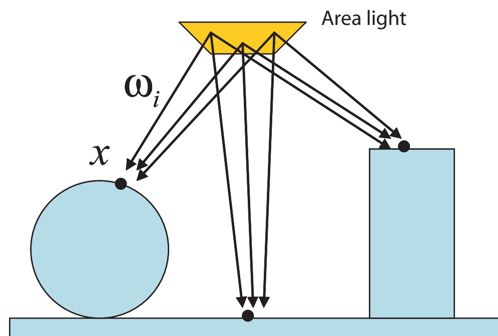


Figure 2.5: Area light source.

2.2.3 Area Light Source

Perhaps the type of light source that resembles the ones found in real life the most is the area light source. An area light source is defined as a full fledged 3D object, just like any other in the scene. It emits light over its entire surface area, therefore illuminating a particular point in the scene from multiple directions, unlike the point and directional light sources in which a point is lit only from one direction (Figure 2.5). The rendering equation can be adjusted to accommodate area lights

by changing the integral over solid angle to integral over the area, A, of the light source. The derivation, starting from the original rendering equation, is as follows.

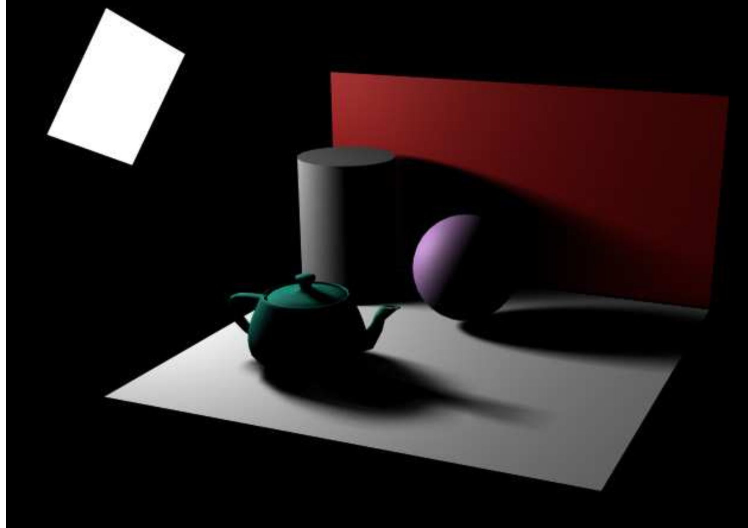


Figure 2.6: Soft shadows due to area light source illumination.

$$L(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega L(x,\omega_i) f_r(x,\omega_i,\omega_o)(\omega_i \cdot n_x) d\omega_i \tag{2.18}$$

$$\text{using definition: } d\omega_i = \frac{dA_z cos\theta_z}{||x-z||^2} = \frac{dA_z(-\omega_i \cdot n_z)}{||x-z||^2}, \text{ where } \omega_i = \widehat{z-x}, \text{ and } z\varepsilon A \tag{2.19}$$

$$L(x,\omega_o) = L_e(x,\omega_o) + \int_A L(x,\omega_i) f_r(x,\omega_i,\omega_o)(\omega_i \cdot n_x) \frac{(-\omega_i \cdot n_z)}{||x-z||^2} dA_z \tag{2.20}$$

The computation involves sampling the area of the light source at points z and evaluating the integral using those points. In situations where a point in the scene is lit from multiple incoming directions, such as in the case of area lights, light from some directions can be occluded. Therefore, unlike the case of single direction illumination in which a point is either completely shadowed or

completely lit, in multiple direction illumination a point can be *partially* in shadow. The shadows produced in this case are known as *soft shadows* (Figure 2.6).

It is important to note that due to the integral evaluation in the rendering equation, area light sources are seldom used in real-time applications. Instead, point and directional lights are employed since the computation of the outgoing radiance is much simpler.

We now shift our focus to the second category of light sources, the image-based lights. In these representations, the lighting function is defined using images instead of geometric primitives. In the following, we describe two different types of image-based lighting techniques: environment lighting and light maps.

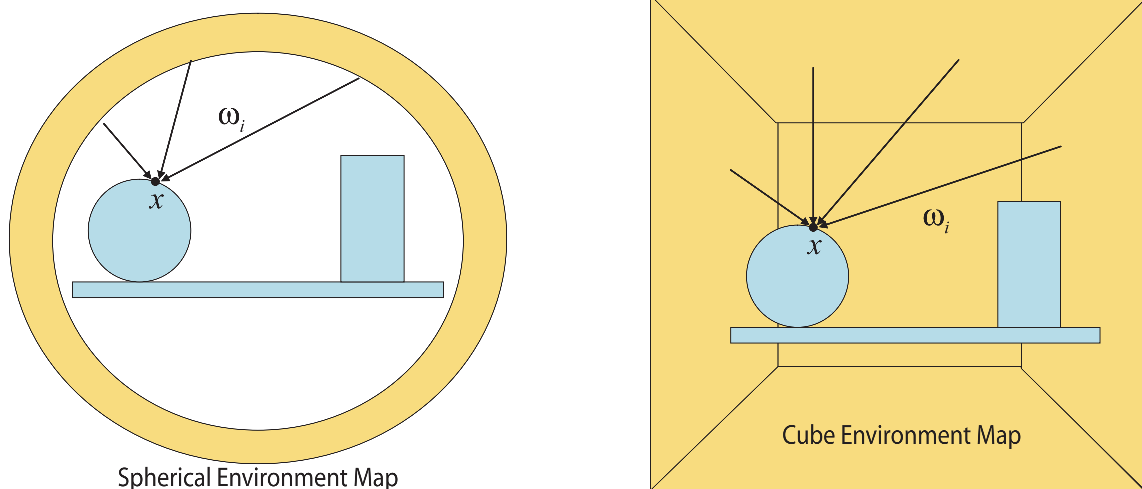


Figure 2.7: Environment lighting representations: (a) Spherical Environment Map, and (b) Cube Environment Map.

Figure 2.8: Real-world environment maps captured at the Grace Cathedral: (a) spherical representation, and (b) cube map cross representation. Images taken from http://www.debevec.org.

2.2.4 Environment Lighting

Environment lighting is one of most popular lighting techniques in computer graphics, and was introduced by Blinn in 1976 [BN76]. Environment lighting is represented using one or more images, known as environment map, that define the incoming light from all possible directions at some reference point. Therefore, the incoming radiance, $L(x, \omega_i)$ can obtained from the environment map by querying it using the direction vector, ω_i. Furthermore, environment lighting assumes that the light is coming from infinitely far away. This means that value of the incoming radiance in direction ω_i at any point in the scene is the same. Therefore, the rendering equation slightly simplifies to:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} L(\omega_i) f_r(x, \omega_i, \omega_o)(\omega_i \cdot n_x) V(x, \omega_i) d\omega_i \tag{2.21}$$

where $L(\omega_i)$ is obtained from the environment map.

Two popular forms of environment maps exist in computer graphics: spherical map (Figure 2.7(a)) and cube map (Figure 2.7(b)). Spherical maps store the environment lighting in a single image, whereby a textured sphere representing the environment is flattened onto the image. On the other hand, cube maps consist of six images, each representing a face of a textured cube representing the environment. In some cases all six faces of the cube are stored in a single image, forming a cross shape as shown in Figure 2.8(b). Cube maps are generally preferred over spherical maps because spherical maps introduce stretching artifacts and distortions near the poles of the sphere, whereas properly filtered and utilized cube maps produce virtually artifact-free rendering.

2.2.5 Light Maps

Light maps are cheap and fast illumination representations, and are often employed in computer games. A light map is an image that stores irradiance at every point in the 3D scene. It essentially provides the total light incident at a given point, and therefore does not have any directionally dependence unlike environment maps. Furthermore, the visibility function is also accounted for directly in the light map and hence does not need to be computed at render stage. Light maps are constructed such that every region in the 3D scene maps to a location in the light map image. During the rendering stage, a point in the scene is shaded by modulating the corresponding light map value with its texture. The rendering equation therefore is quite simple.

$$L(x,\omega_o) = L_e(x,\omega_o) + E(x) f_{hdr}(x,\omega_o); \tag{2.22}$$

where $f_{hdr}(x,\omega_o)$ is known as the *Hemispherical-directional reflectance function*, and is the integral of the BRDF over all illumination directions, ω_i.

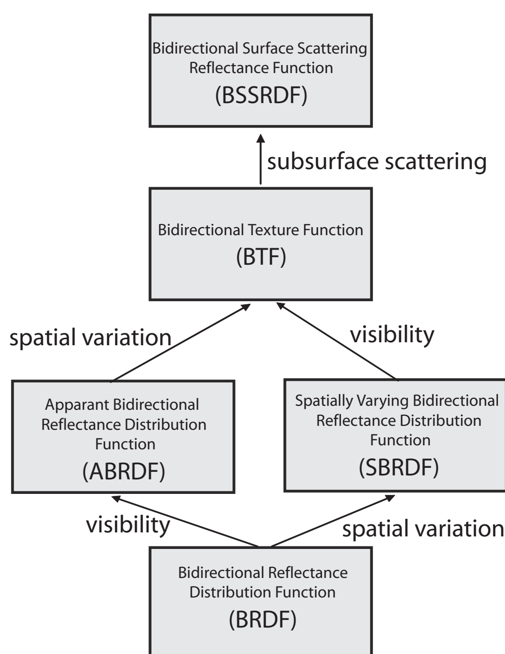


Figure 2.9: Hierarchy of reflectance models.

2.3 Appearance Modeling

Thus far we have discussed the various representations of light sources employed in computer graphics, and the mathematical equations that describe the interaction of light with 3D objects in a

scene. We now describe the modeling of reflectance properties of materials, which give them their unique visual appearance.

Light impinging on the surface of an object undergoes a series of events. A fraction of the light is reflected directly off the surface, whereas the rest penetrates the surface and gets transmitted inside the material. As light travels inside the material, it gets absorbed and scattered. Eventually, the light is either completely absorbed, or some of it reemerges from the surface, perhaps at a different location from where it entered. For rendering purposes, we are ultimately concerned with the distribution of the light field that is reflected from the surface of the object, given the incident light field. This distribution is modeled with reflectance functions, f_r, which we introduced earlier in this chapter in the context of the rendering equation. Researchers have developed an entire hierarchy of reflectance functions ranging from the very general models that can be used to convincingly represent most real-world materials, to very simplistic ones that are faster to evaluate but do not faithfully represent many, if any at all, real-world materials. This hierarchy is depicted in Figure 2.9. We will begin at the bottom of the hierarchy by looking at the BRDF (Bidirectional Reflectance Distribution Function) and work our way up.

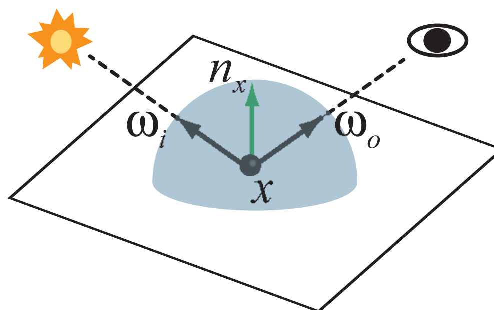


Figure 2.10: The Bidirectional Reflectance Distribution Function.

2.3.1 Bidirectional Reflectance Distribution Function

The BRDF is a 4-dimensional function that relates the outgoing radiance to the irradiance at a specific a point on a surface (Figure 2.10). It is defined as the ratio of the reflected radiance in direction, ω_o, to the irradiance incident from direction, ω_i, at a point on a surface:

$$f_r(\omega_i, \omega_o) = \frac{dL(\omega_o)}{dE(\omega_i)} = \frac{dL(\omega_o)}{L(\omega_i) cos\theta d\omega_i} \tag{2.23}$$

The BRDF is considered 4-dimensional since the incoming and outgoing directions, ω_i and ω_o, are represented in polar coordinates, (θ_i, ϕ_i) and (θ_o, ϕ_o). Although the BRDF is a ratio, and therefore does not have any physical units, it is often expressed in 1/sr. BRDFs have a few important properties: non-negativity, reciprocity:

$$f_r(\omega_i, \omega_o) = f_r(\omega_o, \omega_i) \tag{2.24}$$

and energy conservation:

$$\int_\Omega f_r(\omega_i, \omega_o)(\omega_i \cdot \omega_o) d\omega_i \leq 1, \text{ for any given } \omega_o \tag{2.25}$$

Therefore, BRDFs do not take on negative values; the range is defined as $[0, \inf)$. Secondly, they are symmetric; that is, if the light and view directions are interchanged, the function returns the

same value. Lastly, BRDFs conserve energy, and therefore the total energy reflected is less than or equal to the total incident energy.

In computer graphics, BRDFs are employed in two forms: as *analytic models* described by mathematical equations, or as *discrete functions* measured and stored in lookup tables. We first present some of the common analytic BRDF models, and then later discuss how BRDFs are measured from real-world materials and used to render virtual objects that visually resemble these materials.

Lambertian Diffuse BRDF: This is the simplest BRDF in computer graphics. It returns a constant value for all input, indicating that the light is reflected equally in all directions.

$$f_r(\omega_i, \omega_o) = f_r^d \tag{2.26}$$

The Lambertian diffuse BRDF is used to render objects with non-shiny, matte appearance.

Phong BRDF: The Phong BRDF is frequently used in computer graphics due to its simplicity and ability to represent both diffuse and specular materials. It is defined as:

$$f_r(\omega_i, \omega_o) = k_d + k_s cos^s \theta_r \tag{2.27}$$

where k_d and k_s are the diffuse and specular coefficients, θ_r is the angle between the view direction, ω_o, and the reflected light direction, $\omega_r = reflect(\omega_i, n_x)$, and s is the "shininess" parameter.

The Phong BRDF emulates specularity by principally reflecting light in the mirror reflection direction. Therefore, the reflection appears the brightest, forming a specular highlight, when the view direction is perfectly aligned with the reflection direction, and gets less and less bright as the angle between the directions increases. The falloff is governed by the shininess parameter. A large shininess value produces a sharp falloff indicating a highly specular material, and vice versa. A variation of the Phong BRDF that is often used is the Blinn BRDF. In the Blinn BRDF uses the *halfway* direction between the light and view directions, instead of the mirror reflection direction. It is defined as follows:

$$f_r(\omega_i, \omega_o) = k_d + k_s cos^s \theta_h \tag{2.28}$$

where:

$$cos^s \theta_h = (\omega_h \cdot n_x)^s \tag{2.29}$$

$$\omega_h = \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|} \tag{2.30}$$

This model is sometimes preferred since the computation of the halfway direction is simpler than the mirror reflection direction.

Cook-Torrance BRDF: The Cook-Torrance BRDF is a physically based BRDF that attempts to model the reflectance of a material at the microstructure level of the surface, and serves as a more accurate alternative to the simplistic Phong BRDF. The Cook-Torrance BRDF assumes that the surface of the material is made up of perfectly reflecting microfacets, arranged in V-shaped

grooves. These microfacets act like tiny mirrors oriented in different directions, thus reflecting the incoming light according to their orientation. The Cook-Torrance BRDF is defined as follows:

$$f_r(\omega_i, \omega_o) = \frac{k_d}{\pi} + \frac{k_s}{\pi} \frac{FDG}{cos\theta_i cos\theta_o} \tag{2.31}$$

where F is the *Fresnel reflectance*, and

$$D = \frac{1}{m^2 cos^4\theta} e^{-[\frac{tan\theta}{m}]^2}, \text{m is a material parameter} \tag{2.32}$$

$$G = min\{1, \frac{2cos\theta_h cos\theta_i}{\omega_i \cdot \omega_h}, \frac{2cos\theta_h cos\theta_o}{\omega_i \cdot \omega_h}\} \tag{2.33}$$

D is known as the Beckman Distribution, and it describes the distribution of the microfacets on the surface of the material. G is the shadowing and masking term, which represents the fraction of light that is incident and reflected after being occluded by other microfacets.

Oren-Nayar BRDF: Similar to the Cook-Torrance BRDF, the Oren-Nayar BRDF is also a physically based model that assumes that a material surface is made up of microfacets arranged in V-shaped grooves. However, these microfacets are perfectly Lambertian diffuse, rather than mirrors. The Oren-Nayar BRDF is used to model reflectance of matte surfaces, and therefore serves as a more accurate replacement for the Lambertian Diffuse BRDF.

Although a number of other analytic BRDF models exist, such as the Lafortune BRDF and the Ward Anisotropic BRDF, they are beyond the scope of this thesis. We will instead shift our focus to the second type of BRDF representation, the discrete functions.

Figure 2.11: Screen captures of our BRDF viewer application.

Discrete BRDFs and Measurement: In addition to the analytic form of the BRDF, they are also sometimes stored as discrete values in a tabular form, indexed by the incoming and outgoing directions. This form is generally employed when a BRDF is measured from a real-world material. The process of measuring a BRDF involves taking digital images of a sample of a particular material from different viewing directions and under different illumination directions. A pixel in the image represents a point on the surface of the material, and the value of the pixel in the image taken from a given pair of view and illumination directions, ω_o and ω_i, corresponds to a value of the BRDF, $f_r(\omega_i, \omega_o)$. The entire BRDF can be constructed in this fashion, as densely or sparsely as required. For rendering purposes, these discrete BRDFs can either be *fitted* to an existing analytic model, or can be utilized directly.

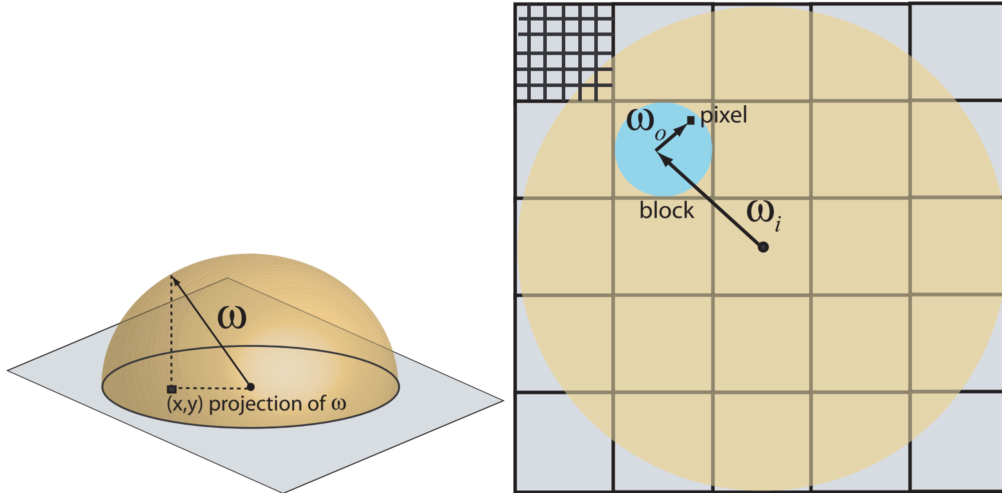


Figure 2.12: Discrete BRDF storage scheme on the GPU. (Left) Projection of a 3D vector onto a 2D plane. (Right) Selecting a pixel location using the view and illumination vectors.

Real-time rendering on the GPU directly using a discrete BRDF involves storing this 4-dimensional function in the video memory of the graphics card. Recall from our earlier discussion that the graphics card allows storage only in the form of 1D, 2D, and 3D textures. Therefore, in order to store the BRDF in video memory, it has to be mapped to a lower dimensional storage representation. We propose a simple projection-based technique for storing discrete BRDFs in 2D texture images. Our storage scheme is depicted in Figure 2.12. The goal is to map a given pair of incoming and outgoing directions, ω_i and ω_o, to a unique pixel in the image so that the corresponding BRDF value can be stored at that pixel. The image is first divided up into a grid of $n \times n$ blocks. Each of these blocks consists of $m \times m$ pixels. We use the incoming direction, ω_i, to select a particular block in the image. This is done by projecting the incoming direction vector onto the plane perpendicular to the normal vector. The projection gives us a 2D location on the plane, which is used to index the corresponding block. Once the block has been selected, we repeat the same projection process with the outgoing direction vector to index the pixel inside the block. Therefore, a unique

pixel can be identified for a given pair of incoming and outgoing directions in this manner. With this mapping, a discrete BRDF can be stored in a single 2D image in video memory, and can be directly used for rendering. We have employed our mapping technique to implement a BRDF viewer, which renders 3D objects under point light illumination using discrete BRDFs. Some images from our implementation are shown in Figure 2.11.

2.3.2 Bidirectional Texture Function

Although the BRDF is able to represent the reflectance properties of certain materials faithfully to a degree, we will now describe how it can be further generalized to be able to represent a larger class of materials.

Thus far we have discussed reflection of light from an incoming direction to an outgoing direction at a single point, as described by the BRDF. When 3D objects are rendered using a BRDF, the same reflectance function is applied at all points over the entire surface of the object. Although this is sufficient for rendering objects that are made of a uniform material, real-world objects seldom have uniform reflectance properties over the entire surface. The reflectance function generally varies from one point on the surface to another. This variation is accounted for in an extended reflectance function known as the *Spatially Varying BRDF* or *SBRDF*. A SBRDF is simply a collection of different regular BRDFs, such that each point on the surface is represented by a separate and

unique BRDF. The SBRDF is therefore a 6-dimensional function: $f_r(x, \omega_i, \omega_o)$, where x represents the 2D coordinates of a point on the surface.

In addition to the spatial variation, the reflectance function of certain materials is also influenced by the occlusion due to variations in the mesostructure of the surface. This results in fine-scale self shadowing, which greatly alters the visual appearance of the material. These materials are modeled using Bidirectional Texture Functions, or BTFs. A BTF is essentially a SBRDF modulated with the occlusion term. It is important to note that this occlusion term is not the same as the visibility function, V, in the rendering equation. V represents occlusion at a macro-scale, that is, occlusion due to other surfaces in the scene. Whereas the occlusion accounted for in BTF is at the meso-scale.

A BTF, which is also a 6-dimensional function like the SBRDF, is generally represented as a set of images. Each image represents the reflectance function under a specific pair of view and illumination directions, and each pixel of the image represents the spatial variation of the reflectance function. This is a kind of image-based representation, since the reflectance function due to complex variations in the mesostructure is represented using images, rather than explicitly modeling the geometry of the surface. BTFs, therefore, are able to reproduce complex optical effects relatively cheaply compared to the geometric modeling approach, which would require a very large number of polygons to accurately model the surface mesostructure, and also incur a very expensive lighting cost. Due to this, BTFs have become quite popular in real-time realistic rendering. Like BRDFs, BTFs from real-world materials can also be acquired. The measurement process is very similar to that of the BRDF, the only difference is that for BTFs we are concerned with

the reflectance over the entire surface of the material rather a single point. Note that when the reflectance information is captured from real-world objects by taking photographs with a digital camera, the meso-scale occlusion is included. Therefore, the set of images represents a BTF, and not a SBRDF. More details regarding BTF acquisition, compression, and rendering are presented in our work on real-time rendering of realistic looking grass in Chapter 4.

2.3.3 Bidirectional Surface Scattering Reflectance Distribution Function

The appearance models that we have discussed so far consider light reflection only at a single point. Even though SBRDFs and BTFs account for spatial variation in the reflectance, they still assume that the light reflected at a given point is only due to the light incident at the same point. However, the real-world interaction of light with materials is more involved. Light incident at a particular point, x_i, on the surface of an object can penetrate the surface and get transmitted inside the material. There, it gets scattered and absorbed, and can eventually reemerge at the surface at a different point, x_o. This process is known as subsurface scattering, and it plays a critical role in the visual appearance of translucent objects such as marble and skin. Light inside translucent materials gets highly scattered, unlike in matte materials where it is quickly absorbed. Due to this property, parts of a translucent object that are not directly lit do not appear completely dark. For example, if a translucent object is lit from the back, an observer looking at the front facing surface can still see a part of the region lit from the scattered light (Figure 2.13).

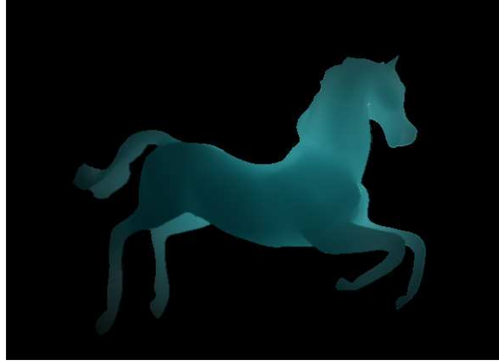


Figure 2.13: Scattering in a back lit 3D object.

Subsurface scattering is represented by the Bidirectional Surface Scattering Reflectance Distribution Function, or BSSRDF. The BSSRDF, $S(x_i, \omega_i, x_o, \omega_o)$, is an 8-dimensional function that relates the incoming light at a point x_i in direction ω_i to the outgoing light at a point x_o in direction ω_o. Therefore, the BSSRDF indicates the amount of light that is scattered from one point to another. In order to render an object with subsurface scattering using the BSSRDF, one must account for the incoming scattered light at a particular point from all other points on the surface. The rendering equation is thus modified to accommodate BSSRDFs as follows:

$$L(x_o, \omega_o) = L_e(x_o, \omega_o) + \int_\Omega \int_A L(x_i, \omega_i) S(x_i, \omega_i, x_o, \omega_o)(\omega_i \cdot n_x) d\omega_i dx_i \tag{2.34}$$

As in the case of BRDFs, BSSRDFs can also be represented using analytic models or measured from real-world materials. Analytic BSSRDF models are generally defined in terms of various physical properties of materials, such as the scattering coefficient, absorption coefficient, refractive index, and phase function. Perhaps the most popular analytic BSSRDF model in computer graphics is the one introduced by Jensen et al., which uses the "dipole source approximation" to

solve for the multiple scattering of light in homogenous translucent materials. It is an elegant and simple solution that models the diffusion of light due subsurface scattering inside a highly scattering material. The diffusion approximation depends only on the distance, thus simplifying the BSSRDF: $S(||x_i - x_o||, \omega_i, \omega_o)$. Although the evaluation of Jensen's BSSRDF is simple, the rendering process still involves computing the integral over the area in order to accumulate all the in-scattered light, which can be challenging for real-time rendering. In Chapter 6, we present an interactive algorithm for rendering translucent objects, which efficiently computes this integral in image-space. We also provide a more detailed description of the dipole BSSRDF model in that chapter.

BSSRDFs can also be acquired from real-world materials using measurement techniques similar to the ones used for BRDFs and BTFs. The BSSRDF is measured by illuminating a sample material at a single, x_i, from a particular direction, ω_i point and taking a photograph from a particular view direction. Each pixel of the image represents a point x_o on the surface of the material, and the value of pixel indicates the amount of light scattered from x_i to x_o, that is reflected in the view direction, ω_o. This process is repeated for varying x_i, ω_i, and ω_o to construct the entire BSSRDF. Measured BSSRDFs are ideal for rendering heterogenous translucent materials since they can account for the nonuniform subsurface scattering properties of materials. However, efficient storage and representation of discrete 8-dimensional BSSRDFs is a challenge, specially for real-time rendering.

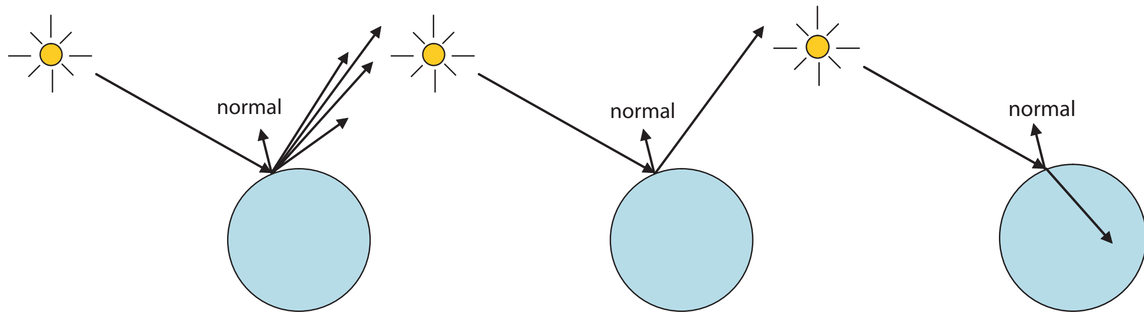


Figure 2.14: Incident light reflected off a (a) shiny surface, and (b) a perfect mirror. Notice that in the case of a perfect mirror, all the light is reflected in a single direction. (c) Light refracted by a perfect refractor.

2.4 Reflection and Refraction

In our discussion on appearance modeling, we looked at a number of different reflectance functions, which are used to describe how light interacts with different materials. These functions essentially describe the angular and spatial distribution of the reflected light field due to some incident light field. For certain materials, the distribution of the reflected light field is quite different and unique, and therefore cannot be represented using general reflectance functions. Perfect *mirror reflectors* and *refractors* are two such materials, which are of special interest in computer graphics due to their unique appearance.

In perfect mirrors, light from a single incoming direction is reflected into a single outgoing direction (Figure 2.14). The relation between the incoming and outgoing directions is described by the *law of reflection*, which states that the angle between the incoming direction and the surface normal at the point is exactly the same as the angle between the outgoing direction and the surface normal. This means that incident light is not dispersed in multiple directions on reflection. Due to this

behavior of perfect mirrors, an observer is able to see a sharp reflection image of the surroundings on the surface of a mirror reflector. In computer graphics, the reflectance of mirrors is represented using a delta function. Therefore, the rendering equation for mirrors is described as:

$$L(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega L(x,\omega_i) f_r(x,\omega_i,\omega_o)(\omega_i \cdot n_x) d\omega_i \tag{2.35}$$

$$f_r(x,\omega_i,\omega_o) = F_r(\omega_o)\frac{\delta(\omega_o - \omega_{refl})}{\omega_i \cdot n_x} \tag{2.36}$$

where F_r is the Fresnel reflectance function and ω_{refl} is the mirror reflection direction of ω_i about n_x.

The interaction of light with refractive materials is similar. As with mirror reflection, the incoming light from a single direction leaves the surface in a single outgoing direction, in this case, the refraction direction, which is described by the *law of refraction*, also known as *Snell's law*. Snell's law states that the ratio of the sines of the angle of incidence (angle between incoming direction and the surface normal) and the angle of refraction (angle between outgoing direction and the surface normal) is a constant, known as the relative refractive index, and depends on the media through which the light is propagating. Refraction refers to the change in the direction of propagation of a wave form that occurs at the interface of two mediums due to change in speed as the wave moves from one medium to another. Light waves undergo this event when they impinge the surface of a refractive object, and then continue propagating inside the object in the refraction direction. A second refraction event takes place when the light exits the object at the surface. In graphics,

refraction is managed the same way as reflection by using a BRDF involving the delta function:

$$f_r(x, \omega_i, \omega_o) = F_t(\omega_o)\frac{\delta(\omega_o - \omega_{refr})}{\omega_i \cdot n_x} \tag{2.37}$$

where F_t is the Fresnel transmittance function and ω_{refr} is the refracted direction of the incoming light direction.

Refraction and reflection are also special because they give rise to other interesting optical phenomena. We will briefly discuss a few of these phenomena next.

2.4.1 Caustics

Light rays reflecting or refracting from curved surfaces, sometimes get focused at small regions. Due to this, the light energy concentration at these particular regions gets much larger than the surrounding regions, and therefore appear much brighter. These bright regions of focused light are known as caustics. Caustics are very frequently observed in the real world, such as the cardioid shaped reflective caustics in coffee mugs and cups, or the underwater refractive caustics in swimming pools. Hence for realistic image synthesis, rendering of caustics in presence of reflective and refractive objects is very important.

Caustics are indirect illumination effects; that is, light interacts with other surfaces before it converges at a particular region to form caustics. As discussed earlier, computing indirect illumination is quite expensive, and is not feasible for real-time rendering in most cases. However, caustics can be isolated from the rest of the indirect illumination and computed independently in an approximative manner to facilitate real-time rendering. In Chapter 5, we present our real-time image-space caustics rendering algorithm that is able to produce visually convincing caustics at interactive frame-rates.

2.4.2 Dispersion

Perhaps the most spectacular and well known optical phenomena is the chromatic dispersion of light into its spectral components, which is observed in rainbows. This dispersion occurs when light travels from one medium to another, since the speed of light waves depends on the wavelength. Therefore, when light gets refracted, each spectral component wave changes its direction of propagation to a different degree, according to their respective changes in speed at the interface of the media. If the discrepancy is large enough, the spectral components appear separated.

In computer graphics, physically based spectral renderers have been developed, and are able to account for chromatic dispersion. For the purposes of real-time rendering, however, this effect can be approximately produced by refracting light using different refractive indices for each of the red,

green, and blue channels. We show results of chromatic dispersion using this technique with our real-time caustics rendering algorithm, explained in Chapter 5.

CHAPTER 3

REAL-TIME RENDERING OF COMPLEX GEOMETRY

Managing geometrically complex surfaces is a prominent challenge in real-time rendering. Most of the interesting objects, natural as well as synthetic, have complex surface structures that require huge sets of polygons for the accurate modeling of their complex structural feature variations such as those on the trunk of a tree. This limitation immensely hinders the process of rendering such objects, and is virtually impossible for applications requiring rendering at interactive frame-rates. However, since these complex structured objects and scenes are quite common and frequently observed in many situations, a number of different solutions have been proposed over the years. One way to handle large 3D meshes is to perform mesh simplification up to the level which can be managed by the current graphics hardware. However, this simplification process generally results in the loss of all interesting surface features, and the object loses its original visual appeal. The surface mesostructure produces some significant visual effects such as fine-scale self shadowing, occlusion and silhouettes, and therefore must be maintained in order to retain the natural look of the object.

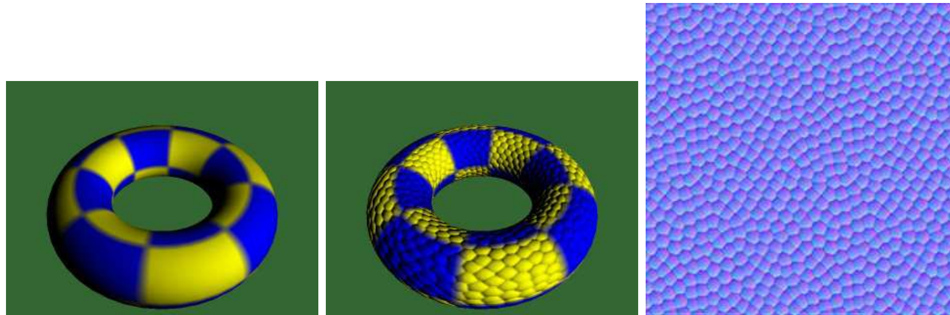


Figure 3.1: A simple torus mesh (left) rendering with bump mapping (center) using a bump map texture (right).

3.1 Bump Mapping and Displacement Mapping

To remedy the problem of dealing with unmanageable amounts of geometry, image-based techniques have been explored. The main advantage of employing an image-based solution is that it decouples the rendering complexity from the geometric complexity of the surface being rendered. One of the earliest and most commonly used image-based techniques is *bump mapping*, which was introduced by Blinn [Bli78]. Bump mapping involves using an image, known as "bump map", to represent the surface normals at each point on a surface. The lighting is then computed at each point by using the surface normal from the bump map at the corresponding texture coordinates. In doing so, a simple low polygon mesh can be shaded such that it appears to have high surface detail (Figure 3.1). Bump mapping does not, however, capture all of the visual effects such as shadowing and occlusion since only the surface normals of the mesh are manipulated. A more accurate technique called *displacement mapping* was introduced by Cook[Coo84]. This method utilizes a height field texture which is used to displace the actual geometry of the surface. Modern graphics hardware allows texture lookups in the vertex processing program and therefore can be used to

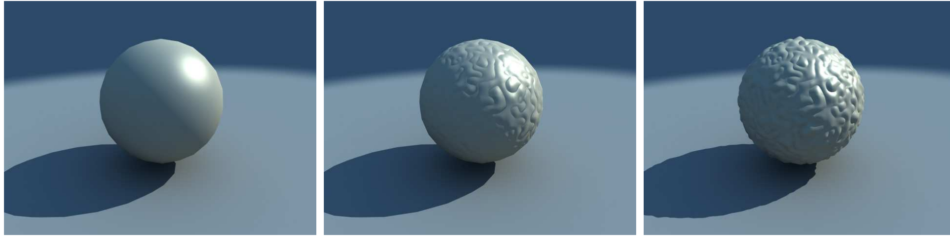


Figure 3.2: A sphere (left) rendering with bump mapping (center), and displacement mapping (right).

access the height field to perform per-vertex displacement. However, the current height field rendering research focus is on per-pixel displacement. This enables strikingly accurate rendering of complex surfaces with fine-scale details. Figure 3.2 shows a comparison of bump mapping and displacement mapping. Notice that unlike the displacement mapped sphere, there are no silhouettes or shadows on the bump mapped sphere.

We propose a new method for performing per-pixel displacement called *Interval Mapping*. Our method, similar to *relief mapping*, performs a linear search to find an interval in which the intersection point occurs, and then we iteratively reduce that interval until the final point is found. Relief mapping, as well as other forms of per-pixel displacement mapping, relies on a binary search to accomplish this. We have discovered that employing the secant root finding method produces superior results. Note that this work was done in collaboration with Eric Risser, and has been published in Journal of Graphics Tools [RSP07].

In the next Section, we present a survey of related previous work in height field rendering.

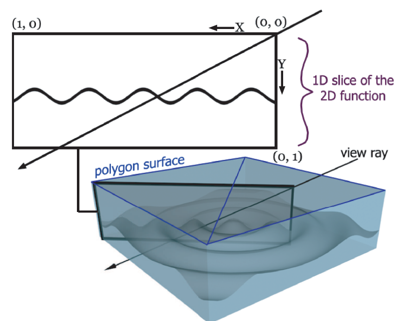


Figure 3.3: 3D height-field and view ray geometry used in Section 3.3 to illustrate the algorithm.

3.2 Background

Parallax mapping [KTI01] is a method for approximating the parallax seen on uneven surfaces. Using the view ray transformed to tangent space, parallax mapping samples a height texture to find the approximate texture coordinate offset that will give the illusion that a three dimensional structure is being rendered. Parallax mapping is a crude approximation of displacement mapping. It can not simulate occlusion, self shadowing or silhouettes. Since it requires only one additional texture read, it is a fast and therefore a relevant approximation for use in video games.

McGuire et al. improved on the algorithm by proposing steep parallax mapping [McG05]. Steep parralax mapping uses a linear search that marches along the view ray checking for intersection with the surface at regular intervals until it finds a point where it is found to have pierced the

surface. The texture coordinates at this point are then used to get an approximation to the actual point of intersection. The need for many texture reads makes steep parallax mapping far slower than parallax mapping. However, the extra testing results in a better approximation that supports self shadowing, occlusion and a visual quality rivaling that of the shells technique without any extra preprocessing or memory requirements other than those of parallax mapping.

View dependent displacement mapping [WWT03] takes a bi-directional texture function (BTF) approach to per-pixel displacement mapping. This approach involves pre-computing for each potential view direction an additional texture map of the same dimensions as the source texture, which stores the distance from the surface of our polygon to the imaginary surface that we want to view. They store a five-dimensional map indexed by three position coordinates and two angular coordinates. The pre-computed data is then compressed and decompressed at runtime on the GPU. This method produces good visual results but requires significant preprocessing and storage to operate.

Per-pixel displacement mapping with distance functions [Donnelly 2005] uses a three dimensional source texture which stores for each voxel the distance to the closest point on the surface to be viewed. The technique is based on sphere tracing [Hart 1996], a technique developed to accelerate ray tracing of implicit surfaces.

Relief mapping [POC05] begins with a linear search in the same fashion as steep parallax mapping. However, once it finds the point at which the ray has pierced the surface, it then performs a binary search along the view ray to home in on the exact point of intersection.

Our interval mapping algorithm is an extension of the relief mapping algorithm. It utilizes an interval reduction technique to converge to the intersection point at a faster rate than the binary search technique used in relief mapping and thus requires fewer iterations for convergence.

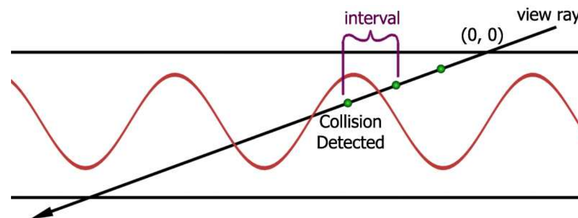


Figure 3.4: Linear search along the view ray.

3.3 The Algorithm

Like similar methods, our interval mapping algorithm conceptually takes a 2D slice out of a 3D height-field as can be seen in Figure 3.3. The algorithm comprises of two main steps:

1. Find initial intersection interval by performing a uniform linear search along the view ray. This involves sampling the view ray at equally spaced steps and looking up the height field until the point on the ray is below the corresponding height value. This step is identical to the first step of the relief mapping technique and is illustrated in Figure 3.4.
2. Iteratively reduce the intersection interval (shown in Figure 3.4) to a specified threshold by adjusting the upper and lower bound of the interval with a ray-line segment intersection test. This step is explained in detail in the remaining paragraphs of this section.

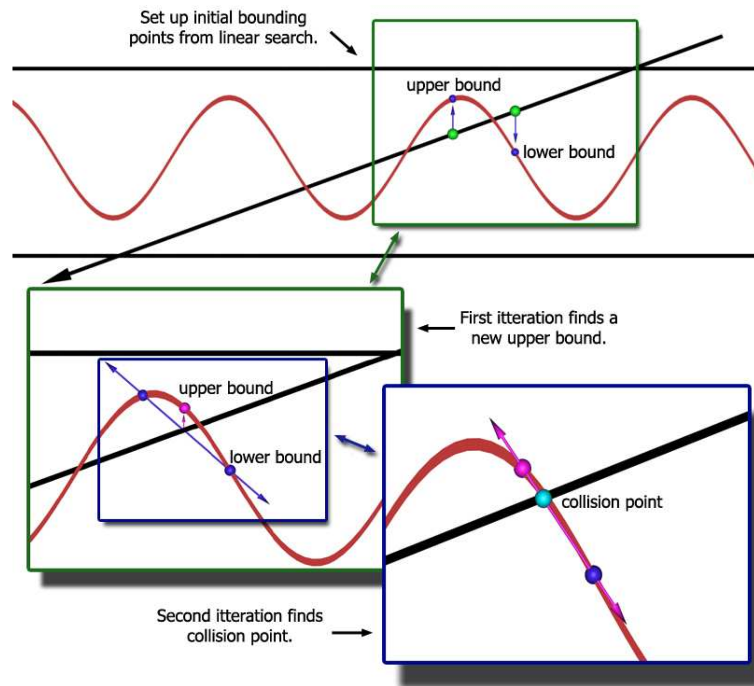


Figure 3.5: Interval search along the view ray; for this specific example an intersection is found in two iterations. The number of iterations for any given case depends on the angle of the incoming view ray and the function it intersects with.

We refer to the two bounding points of the interval as upper bound and lower bound, the upper being the point on the view ray before the intersection and the lower being the point after intersection. Since we are dealing with 2D slices of 3D space, the bounding points are represented by 2D co-ordinates (x,y).

We sample the height field surface at our upper and lower points of the interval to find the two surface points and join the resulting two points to define a line (see Figure 3.5 in which the blue line joins the two points). The point at which this line intersects the view ray is defined as either the new upper or new lower bound depending on whether the height of the surface at that point

is greater or less than our intersection point on the view ray (Figure 3.5 shows the updated upper bound). Our view ray is a line passing through the origin and hence may be represented as

$$A_{view}x + B_{view}y = 0 \tag{3.1}$$

and the line passing through the two surface points may be represented as

$$A_{line}x + B_{line}y + C_{line} = 0 \tag{3.2}$$

where

$$A_{line} = y_{UpperBound} - y_{LowerBound} \tag{3.3}$$

$$B_{line} = x_{LowerBound} - x_{UpperBound} \tag{3.4}$$

$$C_{line} = -B_{line}y_{UpperBound} - A_{line}x_{UpperBound} \tag{3.5}$$

Solving for the point of intersection we get

$$x = C_{line} \frac{B_{view}}{A_{view}B_{line} - A_{line}B_{view}} \quad (3.6)$$

$$y = -C_{line} \frac{A_{view}}{A_{view}B_{line} - A_{line}B_{view}} \quad (3.7)$$

The intersection point updates one of the boundary points of the interval. By iterating through this method, we quickly converge to the point of intersection.

In the next sections we show the results from our methods and compare its performance with that of the relief method.

3.4 Results and Analysis

We implemented interval mapping and relief mapping in GPU using DirectX and HLSL. Both use the same linear search function to find the initial interval, and have the same basic structure for the interval/binary search. The only difference between the two is how they refine the bound.

For relief mapping computation we assigned the midpoint between upper and lower bound i.e (upper + lower)/2 as one of the bounds. For our interval mapping method we set the intersection point as one of the bounds. So the additional cost for interval mapping is an extra division, multiplication and three subtractions per iteration. Interval mapping recovers this loss by converging faster to the

intersection and thus requiring less iterations and fewer slow dependent texture reads. The illustration on the first page demonstrates the faster convergence of our method using a side-by-side comparison of the images generated after two iterations of our method and relief mapping method. Both the methods used five linear searches before carrying out the iterations. In addition to this, we demonstrate the performance increase by counting the number of iterations required by each pixel to reach convergence in each of the methods. We show the comparison results in Figure 3.7.

Empirical results for three sample height field displacements are shown in Figure 3.7. Each row in the figure corresponds to a new height field. The image on the left is the rendered image. The two gray scale images following the rendered images show the number of iterations to convergence per pixel of the image for our interval mapping method and for the relief mapping method. Darker pixels correspond to a smaller number of iterations and brighter pixels correspond to a larger number of iterations. The minimum number of iterations is 1. The maximum number of iterations is arbitrarily set to 10. The performance images clearly illustrate the fact that our algorithm converges much faster than the binary search based relief mapping methods.

Interval mapping uses a similar but refined version of the secant method for root finding [William et al. 2002] of an arbitrary function defined by our height field. The secant method searches for the roots of this function by finding the intersection of the function with the horizontal axis. We can treat our view vector as the axis and attempt to find our root (the intersection point). Interval mapping returns the intersection point with the same running time as the secant method. Mathematically the secant method is defined as

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})} \tag{3.8}$$

where our upper and lower bounds can be thought of as

$$UpperBound = (x_{n-1}, f(x_{n-1})) \tag{3.9}$$

$$LowerBound = (x_{n-2}, f(x_{n-2})) \tag{3.10}$$

In the secant method our two points are the previous two points we've found in our sequence of points. Interval mapping however uses a bounding system to always force convergence, producing slightly better results. If we were to define our view vector as the horizontal line (used in the secant method) then for Equation 3.1:

$$A_{view} = 0 \quad B_{view} = 1 \tag{3.11}$$

Then our final equation for x using the Interval Mapping approach is defined as

$$x = C_{line} \frac{B_{view}}{A_{view}B_{line} - A_{line}B_{view}} = -\frac{C_{line}}{A_{line}} \tag{3.12}$$

The coefficients of the line between UpperBound and LowerBound are obtained from Equations 3.3 to 3.5

$$A_{line} = f(x_{n-1}) - f(x_{n-2}) \tag{3.13}$$

$$B_{line} = x_{n-2} - x_{n-1} \tag{3.14}$$

$$C_{line} = -f(x_{n-1})(x_{n-2} - x_{n-1}) - x_{n-1}(f(x_{n-1}) - f(x_{n-2})) \tag{3.15}$$

$$\tag{3.16}$$

therefore,

$$x = \frac{x_{n-1}(f(x_{n-1}) - f(x_{n-2})) - f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})} \tag{3.17}$$

$$= x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})} \tag{3.18}$$

Which we can see is in the same form as the original equation given for finding the next point using the secant method. It should now be clear that the two methods share running times. This paper does not attempt to improve upon the linear search portion of per pixel displacement mapping as there is already very impressive research going on to speed up this bottleneck in the algorithm.

Efficient Empty Space Skipping for Per-Pixel Displacement Mapping [Kolb 2005] has offered a method for using bounding textures to skip the linear search altogether, offering impressive results. However, when it comes to locating the precise point of intersection, a binary search is used in this technique as well, this approach can be improved with interval mapping.

Additionally, interval mapping can handle self shadowing, the correct depth sorting of geometry with a displacement mapped polygon, and all other effects possible with other per-pixel displacement mapping techniques such as relief mapping. Figure 3.6 demonstrates interval mapped arbitrary polygonal surfaces.

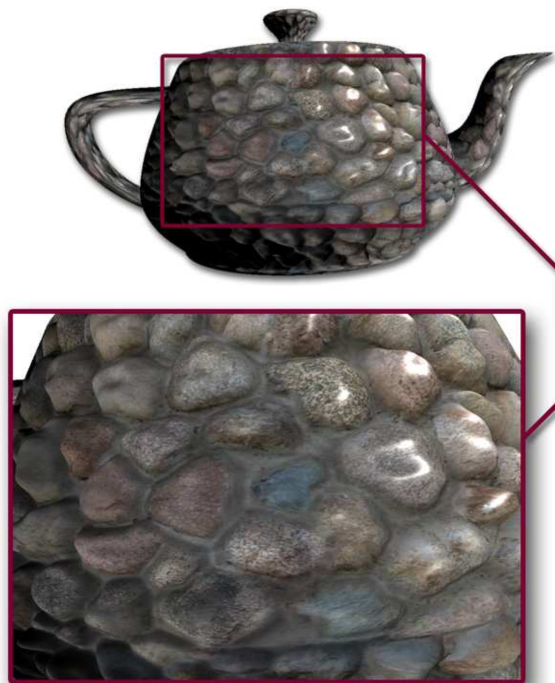


Figure 3.6: Interval mapping over an arbitrary polygon surface.

3.5 Discussion

We have presented a fast converging algorithm based on relief mapping for per-pixel height field rendering. By utilizing the available height field and viewing information, we have shown an efficient way of adaptively reducing the search space for finding the intersection point. Though, empirical evidence shows that both methods require about the same iterations in the worst case, on average the interval mapping algorithm converges in two to three iterations of our intersection interval minimization step whereas it takes five to eight binary search steps.

```
////////////////////////////////////////////////////////////////////
// interval search                                                //
////////////////////////////////////////////////////////////////////
// this portion of code requires a linear search to first be      //
// performed, with the two points right before and right after    //
// collision stored as the upper and lower variables              //
////////////////////////////////////////////////////////////////////

pixel_color.a = 1;
float int_depth = 0;

for(int i = 0; (i < 10) && (abs(pixel_color.a - int_depth) > .01); i++)
{
       float line_slope = (upper_h - lower_h)/(upper_d - lower_d);
       float line_inter = upper_h - line_slope*upper_d;

       float dem = view_slope - line_slope;
       float inter_pt = line_inter / dem;

       tex_coords_offset2D = inter_pt * float2(view_vec.y, -view_vec.x);
       int_depth = view_slope*inter_pt;

       pixel_color=tex2D(heightSampler,(tex_coords_offset2D)+input.tex_coords);

       if(pixel_color.a < int_depth) //new upper bound
       {
              upper_h = pixel_color.a;
              upper_d = inter_pt;
              best_depth = upper_h;
       }
       else //new lower bound
       {
              lower_h = pixel_color.a;
              lower_d = inter_pt;
              best_depth = lower_h;

        }

}
// compute our final texture offset
tex_coords_offset2D = ((1.0f/view_slope)*best_depth)*float2(view_vec.y,-view_vec.x);
```

```
// store interval mapped pixel color

pixel_color = tex2D(textureSampler, tex_coords_offset2D+input.tex_coords);
```

Interval Mapping shader code

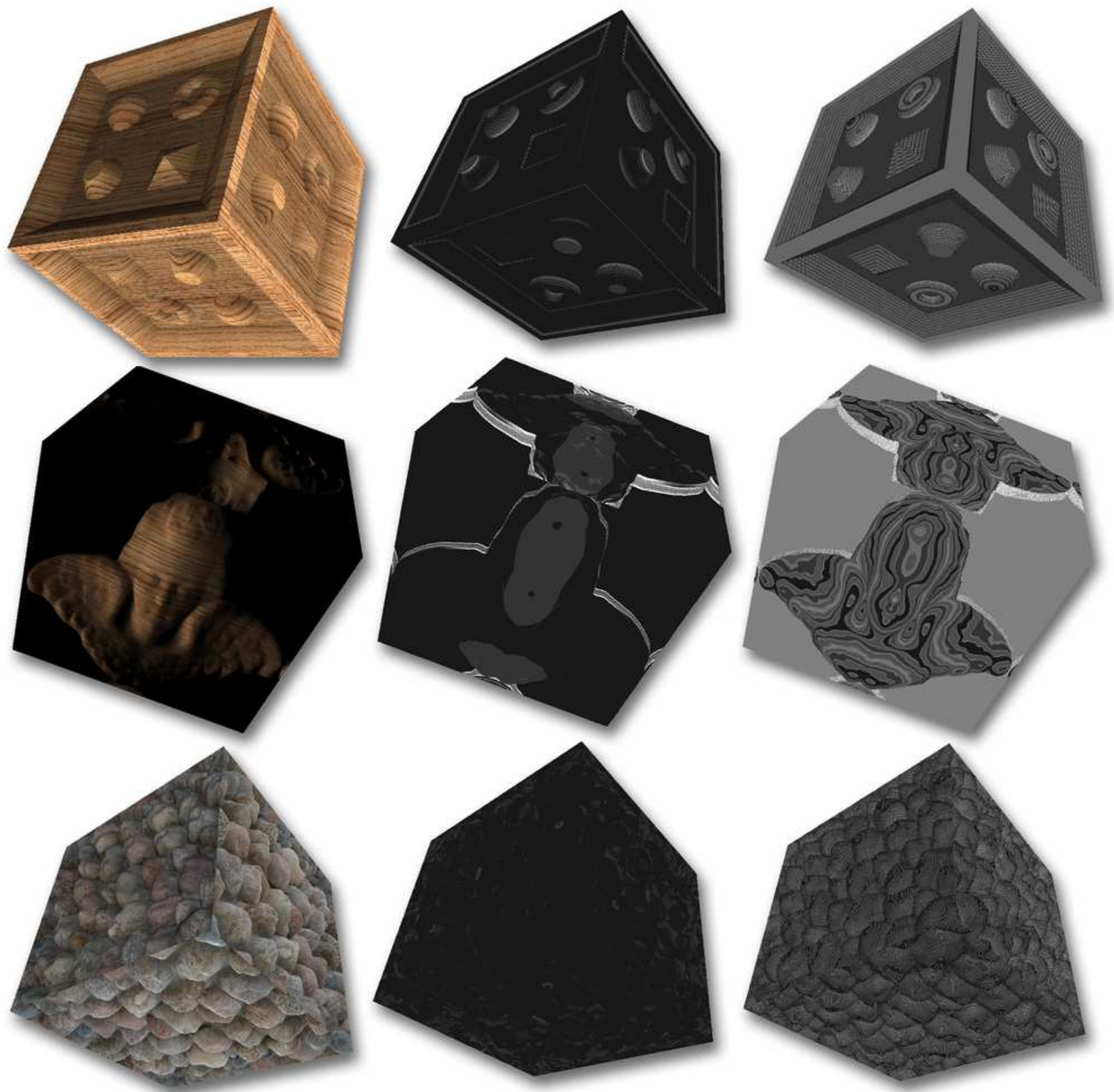


Figure 3.7: Empirical results" Images in the first column are rendered normally using interval mapping. Images in the second and third column show the number of pixel iterations required for convergence. The grey levels in these two columns are coded black for 1 iteration, up to white for a maximum of 10 iterations. The second column shows the results from interval mapping while the third column represents relief mapping. By comparing the average color value of the second column with the third, it is clear to see that interval mapping converges quicker.

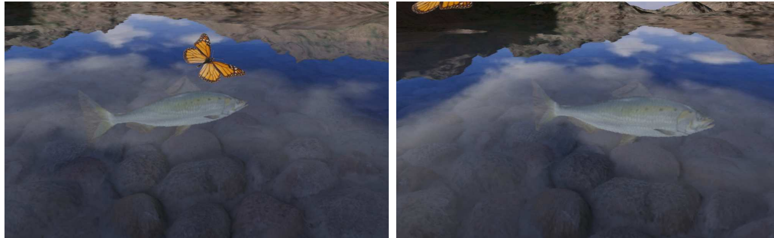


Figure 3.8: Miscellaneous screenshots from our nature demo. The rocks at the bottom of the pond are represented by a single interval mapped polygon.

CHAPTER 4

LIGHTING IN LARGE, COMPLEX SCENES: REAL-TIME RENDERING OF REALISTIC LOOKING GRASS

In the previous chapter, we showed that special images, known as height maps, can be used to represent complex surface structures instead of explicitly modeling them with geometry. We now address the problem of computing lighting in large, complex scenes in the context of rendering terrains of grass. In this chapter, we present our real-time grass rendering algorithm and demonstrate how an image-based approach can be employed to render large amounts of geometry with complete lighting in constant time.

4.1 Introduction

Grass and other forms of natural vegetation occur quite frequently in the outdoors; they are a common sight in sports games and virtual walkthroughs. Although the shape of individual grass blades is not very complex and can be represented with just a handful of polygons, the blades almost never exist in a sparse arrangement. Rather, they span entire fields in huge numbers and thus global illumination computation

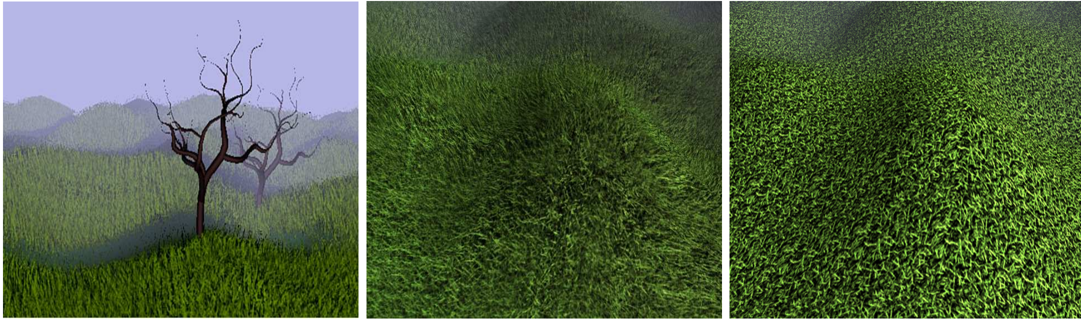


Figure 4.1: Grass rendering with silhouettes using our algorithm (left), and comparison of our method (center) to static texture mapping (right)

becomes impractical for real-time applications. Furthermore, grass blades have complex structural feature variations which give them their characteristic reflection properties. It is thus critical to account for the fine-scale details of the mesostructure in order to produce realistic renderings.

In the context of conventional rasterization algorithms, semi-accurate rendering of grass is possible but it requires massive amounts of geometry. Better visual results can be achieved by using advanced ray tracing systems. However, both approaches are not feasible for real-time applications due to the computational time involved. It is worth noting that the computational time complexity of rendering geometry based algorithms is exponentially proportional to the number of polygons due to the increased complexity of light transport. Image-based solutions, on the other hand, decouple the render time from the geometric complexity of the scene being rendered. Static textures are the most primitive type of image-based rendering and they have been widely employed in games and other real-time applications to render complex surfaces such as wood, sand, and even grass. Although static texture mapping produces significantly realistic results, there are two fundamental issues: (i) the textures are static, hence they are invariant under varying viewing and illumination. This means that, for example, a patch of grass would look exactly the same from every viewing angle. Furthermore, the appearance of the grass would not change under varying illumination; that is, the

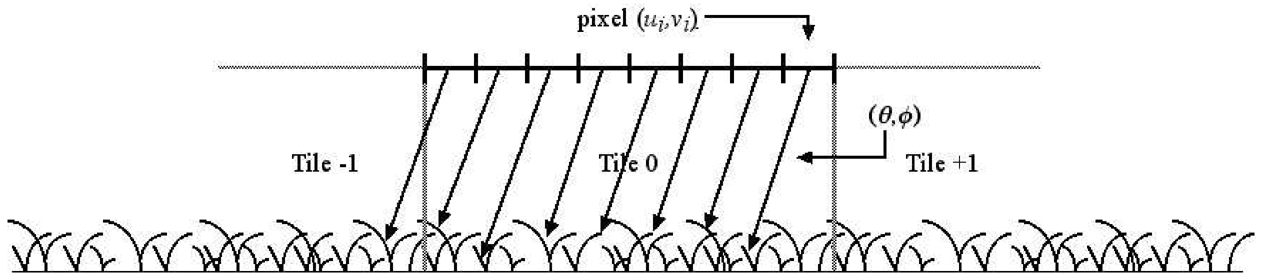


Figure 4.2: BTF dataset acquisition. The image plane is placed directly over a tiling of the grass mesh. The mesh is tiled to prevent discontinuities at the edges. Camera rays are shot through each pixel in parallel based on the current viewing angle being sampled. The intersection distances along the viewing rays are stored in a depth map which is used for silhouette and external occlusion determination.

shadows will remained fixed, the specular highlights will not move, etc. The static nature of the texture map therefore greatly deteriorates the overall visual appeal. The second problem with this technique is (ii) the inherent flatness of texture mapping. Since the texture images are pasted on top of simple, flat polygons, the realism of a 3D environment is lost. Silhouettes can not be achieved, and interaction with external occluders, such as a football placed in a field of grass, is not possible.

This work attempts to overcome the problems of conventional static texture mapping discussed above, while still maintaining the benefits of an image-based method. Hence, our rendering algorithm has three main objectives: (i) produce realistic looking grass at interactive frame rates, (ii) appearance of grass should vary appropriately with changing view and illumination, and (iii) silhouettes and external occlusion should be supported. Our algorithm is an image-based technique which employs a bidirectional texture function (BTF) for grass rendering. We describe the BTF and the algorithm in Section 4.3, following a short survey of related work done in the real-time rendering of natural vegetation in Section 4.2. We present our results in Section 4.4 and conclude with a short discussion regarding future work in Section 4.5.

4.2 Related Work

Although natural vegetation rendering has rallied a fair amount of research activity, the quest for improved visual realism and faster render times continues to draw attention. In this section, we present an overview of the main paradigms employed in the rendering of various plants and trees, and discuss some recent works proposed in this area.

The realistic appearance of grass, or any other form of natural vegetation, depends on two aspects: precise modeling of the shape, and accurate lighting of the model. Grass blades can be modeled by a 3D artist using a modeling package such as Maya or 3D Studio. It is an extremely detail-oriented task requiring an immense amount of manual labor, but the end result is the most precise representation of grass geometry. However, the task of accurately illuminating and rendering this enormous collection of polygons is impractical for real-time applications and therefore a balanced tradeoff between visual realism and rendering cost must be observed.

Employing a particle system was one of the first techniques proposed to break free from the barrier of rendering objects with complex geometry. In [Ree83], Reeves introduced particle systems as a method for modeling fuzzy objects such as fire, smoke, and grass. In this system, trajectory of the particles determines the shape of the grass blade. Although visual detail is lacking, it is able to approximate grass fairly reasonably when viewed from a distance. Furthermore, animation is also easily achievable.

Another technique for modeling a grass blade is to fit an analytic function to its shape. Cubic curves, such as those given by Hermite interpolation, can well approximate the overall shape of the blade. This representation is extremely flexible since the parameters of the curve, such as control points and tangents,

can be easily adjusted to give various shapes. The methods discussed so far have not addressed the issue of accurate illumination. In fact, most of the geometry based rendering algorithms only approximate the illumination by simple diffuse and specular reflection, ignoring the more complex light-surface interactions. The latter gives rise to significant visual effects such as fine-scale self shadowing, sub-surface scattering, and inter-reflection. We thus turn our attention to image-based rendering solutions which are capable of capturing all the intricate visual effects without incurring a huge computational cost.

Elevating from simple texture mapping is the billboarding technique. A billboard is a vertically oriented polygon orthogonal to the view direction and covered with a semi-transparent texture. The transparency in the texture enables it to represent objects with complex shapes such as plants and trees. In [Pel04], two perpendicular polygons containing a section of grass are combined to render large fields of grass with wind animation. Jakulin extends this method for rendering trees by precomputing a different billboard from more than two views [Jak00]. At runtime, the closest pair of billboards is selected and blended together using linear interpolation. Because multiple billboards are also used along the viewing ray, parallax is preserved. Unfortunately, change in illumination is not accounted for in either of the works.

By increasing the number of views and incorporating change in illumination direction, the data structure begins to resemble a Bidirectional Texture Function (BTF). Meyer, et al. employ a hierarchical BTF for rendering trees [MNP01]. A BTF is defined at different levels such as leaves, small branches, main branches, and trunk. At render stage, the tree is represented either using a single BTF or a collection of its component BTFs, based on a level of detail algorithm.

A transition from conventional 2D textures was caused by the introduction of volumetric textures. Decaudin in [DN04] assembles texture slices to form a volumetric dataset of a small section of forest at different

locations and composites them together using traditional volume rendering methods. A forest is rendered in real-time by using different slicing methods based on camera tilt and adaptively varying the slice count using an LOD alogirthm, as well as employing an aperiodic tiling scheme of the volume dataset. Illumination is performed using a fixed directional light source. Although it is desirable to include varying illumination, the size of the dataset becomes very large and prevents the system from doing so.

A hybrid scheme was proposed by Perbet in [PC01] in which a three-level scheme employs geometry close to the eye, volumetric textures at mid-range and a flat texture map at extreme distances.

4.3 Our Rendering System

Our rendering system can be broken down into three main processes: (a) BTF acquisition, (b) compression, and (c) hardware rendering. They are explained in the subsequent sections.

4.3.1 BTF Acquisition

Bidirectional Texture Functions are 6 dimensional functions of position, illumination direction, and view direction.

$$F = F(x, y, \theta_i, \phi_i, \theta_v, \phi_v)$$

BTFs are generally acquired by taking images (accounting for the spatial vartion) of material samples using a digital camera from various viewing directions and under changing illumination [DGN99],[SSK03]. Alternatively, BTFs can be synthesized by creating high quality renderings of virtual 3D models using an offline global illumination renderer [SVL03]. Synthetic BTFs constructed in this manner are more controlled and flexible than those composed of captured natural images since the desired visual effects can be handpicked for inclusion, whereas it is virtually impossible to do so in the case of photographed images. Furthermore, synthetic BTFs are less prone to noise and measurement errors compared to captured BTFs. On the other hand, the realistic appearance of natural images is typically superior to rendered ones.

Our approach for BTF acquisition follows the method proposed by Suykens et al. in [SVL03]. We construct a synthetic BTF dataset for a patch of grass by rendering it under varying view and illumination conditions. For each "image", the view and illumination directions for each pixel in its local coordinate frame are the same, as illustrated in Figure 4.2. The 2D images are then reshaped into 1D vectors and arranged columnwise in the BTF matrix, F. Beside the rendered images, the depth for each pixel along the viewing ray is also stored in a separate texture (Figure 4.2). This depth is the distance between the viewing point and the intersection point of the viewing ray with the grass mesh. If there is no intersection, or if the intersection occurs outside the grass tile being imaged, then a sentinel value such as -1 is stored as the depth. Note that the depth map is only view dependent and thus requires only a few megabytes for storage (approximately 2MB in our implementation). Therefore, we store the depth map in its original form without performing any compression on it since the storage overhead is already small and compression will most likely introduce artifacts. Our rendering algorithm employs this depth map to obtain grass silhouettes, closely resembling the view-dependent displacement map (VDM) suggested by Wang et al. in [WWT03]. We further extend

Figure 4.3: Example of the grass mesostructure influencing the silhouette of the terrain along hill crests.

the application of the depth map by supporting external occlusion. The details of the rendering algorithm are discussed in Section 4.3.3.

4.3.2 BTF Compression and Storage

In this work, we have utilized BTFs composed of 128 x 128 pixel images from 81 different viewing directions and 21 illumination directions. The amount of storage required for the dataset is approximately 80 MB. Current graphics hardware have fairly limited amount of video memory, and thus storing the entire BTF is not feasible. Therefore, compression must be performed on the data in order for it to comfortably fit into video memory. This is essential because locking and filling of video memory is an extremely expensive process and should be avoided whenever possible. We employ Principal Component Analysis (PCA) to compress the BTF dataset. The effectiveness of this compression method depends on the coherence amongst

the constituent images of the BTF. Since the color of all the images in the grass BTF is predominantly green, the chromaticity information is vastly coherent and thus can be highly compressed. This motivates the need for transforming the BTF prior to compression from its original RGB representation into a decorrelated color space such as YCbCr in which the luminance and chromaticity information are decoupled. Next, the BTF matrix, F, is first means-adjusted, and then decomposed using Singular Value Decomposition (SVD).

$$F = USV^T \tag{4.1}$$

The column-space matrix, U, resultant from the decomposition consists of a set of eigenvectors (also known as eigen-textures) sorted in decreasing order of significance. Data compression is obtained by keeping only the first n significant eigenvectors and discarding the rest. This new, lower-dimensional version of matrix U is then used to transform the BTF data into the basis defined by U:

$$F' = U^T F \tag{4.2}$$

Notice that F' is much smaller than F due to the truncation of matrix U. The set of eigenvectors, U, and the weights, F', are stored in video memory as 3D textures indexed by pixel position, view, and illumination directions. Since U is an orthogonal matrix, F can be reconstructed from F and U as:

$$F = UF' \tag{4.3}$$

Reconstruction of the BTF is a simple weighted summation operation which we perform in a fragment shader as explained in the rendering section.

Table 4.1: BTF reconstruction parameters

parameter	description
x,y	2D image coordinates
θ_i,ϕ_i	current illumination direction
θ_v,ϕ_v	current view direction
θ_i',ϕ_i'	illumination direction in local coordinate frame
θ_v',ϕ_v'	view direction in local coordinate frame
F	BTF Matrix
U	Eigenvectors of F
F'	coefficients of F in U
c	number of eigenvectors

As suspected, a great deal of compression was achieved in the chromaticity channels of the BTF dataset. Only the first five eigenvectors were sufficient to accurately reconstruct the original color information. The bulk of the storage and rendering resources were used for the luminance channel which does not compress as much due to the high frequency variations.

4.3.3 BTF Rendering

We perform per pixel reconstruction of the BTF on the GPU using a fragment shader and map the resultant texture onto a low polygon reference surface such as terrain mesh. For every pixel, the current view and illumination directions must first be transformed into the local coordinate frame of the corresponding point on the mesh surface. For efficiency, this transformation can be performed per vertex in the vertex shader program and then interpolated for the in-between pixels. Table 1 gives an overview of the parameters involved in the reconstruction.

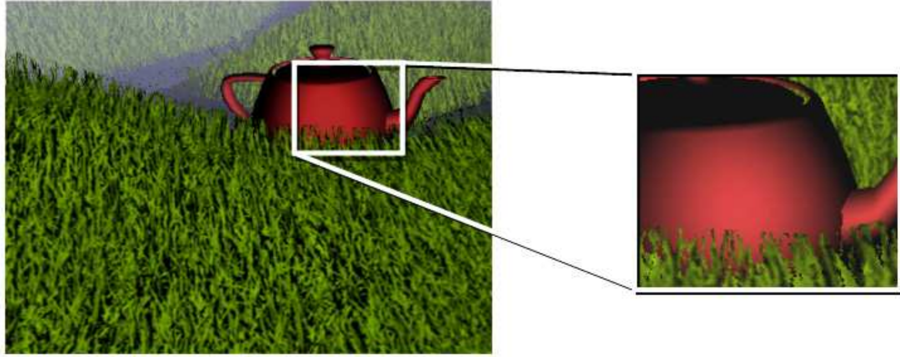


Figure 4.4: Individual grass blades occluding external objects in the scene. (Right) Detailed of the external occlusion effect.

Note that each reconstructed pixel also has a texture coordinate associated with it which determines the position on an individual BTF image. The entire reference surface is covered by tiling the BTF images over it. To reduce discontinuity artifacts along the boundaries of the tiles, bilinear interpolation of final pixel color is performed using the neighboring tiles. The rendering algorithm proceeds as follows. First, the color of the pixel being reconstructed must be computed from the compressed BTF. The reconstruction of a single pixel is given by the following formula:

$$F(x,y,\theta_i',\phi_i',\theta_v',\phi_v') = \sum_{i=1}^{c} U(x,y,i) * F'(i,\theta_i',\phi_i',\theta_v',\phi_v') \tag{4.4}$$

Texture coordinates x,y and the transformed view and illumination directions are used to look up the 3D textures U and F'. Of course, the current directions θ_i',ϕ_i' and θ_v',ϕ_v' will not always match one of the sampled directions. In this case, interpolation is performed using the three closest sampled directions. Note that the reconstruction must be performed for both the luminance and chroma channels, and finally converted into RGB space for output. The next step is to determine silhouettes using the depth map constructed during the BTF acquisition stage. If depth(x,y,θ_v',ϕ_v') = -1, it indicates a silhouette and thus the pixel must be clipped.

This is done by setting the Z-buffer value of the pixel to a large number to ensure that it is not visible. Grass silhouettes using this scheme can be seen in Figure 4.3. It is worth mentioning that the BTF and the depth map are independent entities. Therefore, the resolution of the depth map can be different than that of the BTF images. This enables us to use a higher resolution depth map for better precision.

We enhance our rendering system by adding support for external occluding objects, such rocks and stones, using the depth map. Along with silhouettes, external occlusion gives a 3D appearance to the flat texture map. External occlusion is accomplished by first rendering all the objects except the grass terrain. Then, without clearing the Z-buffer, the grass terrain mesh is rendered. After the pixel color has been calculated, the current pixel depth of the terrain surface is added to the corresponding value retrieved from the depth map and outputted as the final depth value for the pixel. Therefore, if this depth value is greater than that of an external object rendered in the first pass, it will be occluded. The external occlusion effects can be seen in Figure 4.4.

4.4 Results

Our grass rendering system was implemented on a 2.0 Ghz Pentium IV PC with 512MB of physical memory using Pixel Shader 2.0 and the DirectX API. We were able to achieve frame rates of about 20fps on an ATI Radeon X800 128MB graphics card. Since the BTF reconstruction is performed completely on the GPU, multi-pass rendering was required to overcome the shader program instruction and texture lookup limitations. However, since most of the geometry is captured by the BTF and the proxy geometry on which the BTF is mapped onto is fairly low polygon, this does not cause significant overhead. The BTF dataset

Figure 4.5: (Left to right) Images rendered using 4, 40, and 140 coefficients for the reconstruction of the luminance channel of the BTF.

employed in our implementation consisted of 1701 images (corresponding to 81 views and 21 illumination directions) of resolution 128 x 128 pixels. Our experiments show that this resolution was sufficient to avoid tiling artifacts which get magnified at lower resolutions.

Visually pleasing results were obtained using 6 coefficients for the chroma channels and 140 coefficients for the luminance channel to reconstruct the BTF. Using fewer coefficients causes loss of the high frequency luminance information and results in blurry images. Figure 4.5 shows the effect of different levels of compression in the grass renderings. The reason for the large number of coefficients required for grass is that the BTF captures the light-surface interaction at a larger scale rather than just the surface mesostructure. Therefore, changes in illumination and view cause higher variations in the images compared to those of a sample of wool for example.

4.5 Conclusion and Future Work

Although interactive frame rates are achieved through this algorithm, we would like to pursue ideas which can further enhance the compression and appearance of the grass rendering. In particular, it would be

beneficial to research techniques to exploit the flexibility provided by synthetic BTFs to aid compression. For example, removing shadows from the initial BTF would increase coherence amongst the constituent images by getting rid of some high frequencies and thus enable a higher level of compression. The shadows can then be added at render time by the use of the depth map that we already employ to obtain silhouettes and external occlusion. We are also currently exploring level of detail based reconstruction of the BTF images [MCT05]. When grass is viewed from a distance, the fine-scale details of the structure formed by the overlapping grass blades is not prominently visible and appears to be rather homogeneous. We plan to use this notion to formulate a LoD function which determines the number of basis vectors required for a visually pleasing reconstruction based on the current viewing distance. Furthermore, this idea is well supported by the early Z-rejection feature available on current graphics cards which eliminates unnecessary evaluation of the fragment program hence increasing the fill rate.

CHAPTER 5

CAUSTICS MAPPING: AN IMAGE-SPACE TECHNIQUE FOR REAL-TIME CAUSTICS

5.1 Introduction

Caustics are complex patterns of shimmering light that can be seen on surfaces in presence of reflective or refractive objects, for example those formed on the floor of a swimming pool in sunlight. Caustics occur when light rays from a source, such as the sun, get refracted, or reflected, and converge at a single point on a non-shiny surface. This creates the non-uniform distribution of bright and dark areas. Figure 5.2 shows a photograph of caustics from a glass sphere captured with a digital camera. Caustics are a highly desirable physical phenomenon in computer graphics due to their immersive visual appeal. Some very attractive results have been produced using off-line high quality rendering systems; however, real-time caustics remain open to more practical solutions. Deviating from the conventional geometry-space paradigm, which involves path tracing in a 3D scene, intersection testing, etc, we explore an image-space approach to real-time rendering of caustics. Our algorithm has the simplistic nature of shadow mapping,

yet produces impressive results comparable to those created using off-line rendering. We support fully dynamic geometry, lighting, and viewing direction since there is no pre-computation involved. Furthermore, our technique does not pose any restrictions on rendering of other phenomena, such as shadows, which is the case in some previous work [EAJ05]. Our algorithm runs entirely on the graphics hardware with no computation performed on the CPU. This is an important criterion in certain applications, such as games, in which the CPU is already extensively scheduled for various tasks other than graphics.

The remainder of this paper is organized as follows: a short survey of related work is presented in Section 5.2. Our rendering algorithm is then explained in Section 5.3, followed by Section 5.4 discussing results and limitations. We conclude with a summary of the ideas presented in the paper and provide directions for future research in Section 5.5.

Figure 5.1: Image rendered using the caustics mapping algorithm. This result was obtained using double surface refraction (both for the appearance of the bunny as well as for the caustics) at the rate of 42 fps.

5.2 Previous Work

Although caustics rendering, in general, has been subjected to a fair amount of research, a practical real-time caustics rendering does not exist for everyday applications. In this section, we look at some of the earlier work in offline caustics rendering and recent attempts to achieve caustics at interactive frame-rates.

For a fair amount of computational cost, accurate and extremely beautiful caustics can be produced. Introductory work using backward ray-tracing was proposed by Arvo [Arv86], which was then pursued and extended by a number of researchers. In this method, light rays are traced from the light source into the scene as opposed to conventional ray tracing in which the rays emerge from the eye. Photon mapping, a more flexible framework, was proposed by Jensen [Jen96] which handles caustics in a natural manner on arbitrary geometry and can also support volumetric caustics in participating media [JC98b]. Variants and optimized versions of path tracing algorithms have been presented which utilize CPU clusters [GWS04] and graphics hardware [PDC03]; but the computational cost in time and resources are limiting factors in practical application of these techniques to real-time systems. Wyman et al. [WHS04] rendered caustics at interactive frame-rates using a large shared-memory machine by pre-computing local irradiance in a scene and then sampling caustic information to render nearby surfaces. Such pre-computation steps in algorithms restrict their functionality to domains for which the pre-computation was performed and are unable to support fully dynamic scenes. Our algorithm is also based on the backward ray-tracing idea, however it does not require any pre-computation.

Wand and Straβer [WS03] developed an interactive caustics rendering technique by explicitly sampling points on the caustics-forming object. The receiver geometry is rendered by considering the caustic intensity

Figure 5.2: Photograph of caustics from a spherical glass paper weight using a desk lamp to emulate directional spotlighting. The caustics are formed on rough paper placed underneath the refractive object.

contribution from each of the sample points. The authors presented results using specular caustics-forming objects, but refractive caustics can also be achieved with this technique. However, the explicit sampling hinders the scalability of the algorithm since the amount of computation done is directly proportional to the number of sample points used in rendering caustics.

Perhaps the most prominent caustics are those formed in the presence of water. Therefore, the problem of rendering underwater caustics specifically has received significant attention. In early work, Stam [Sta96] pre-computed underwater caustics textures and mapped them onto objects in the scene. Although this technique is extremely fast, the caustics produced are not correct given the shape of the water surface and the receiver geometry. Trendall and Stewart [TS00] have shown refractive caustics to demonstrate the use of graphics hardware for performing general purpose computations. Their intent was to perform numerical integration which they used to calculate caustic intensities on a flat receiver surface. Their method cannot support arbitrary receiver geometry and also cannot be easily extended to handle shadows.

Beam tracing has been employed to produce more physically accurate underwater optical effects, caustics in particular [HH84, Wat90]. In this technique, a light beam through a polygon of the water surface mesh is traced to the surface of a receiver object, hence projecting the polygon onto the receiver. The energy incident on the water surface polygon is used to compute the caustic intensity at the receiver, taking into account the areas of the surface and projected polygons. The intensity contributions from all the participating polygons are accumulated for the final rendering.

Nishita and Nakamae [NN94] present a model based on beam-tracing for rendering underwater caustics including volumetric caustics. Their idea was implemented on graphics hardware by Iwasaki et al. [IDN02]. In a more recent publication Iwasaki et al. [IDN03] adopt a volume rendering technique in which a volume texture is constructed for receiver objects using a number of image slices containing the projected caustic beams. The case of warped volumes which can occur in beam tracing has not been addressed in either of the above. Ernst et al. [EAJ05] manage this scenario and also present a caustic intensity interpolation scheme to reduce aliasing resulting in smoother caustics. However, their algorithm is unable to obtain shadows since it does not account for visibility. In contrast, our algorithm is able to handle shadows and in general does not impose any restrictions on rendering other phenomena.

Independently and in parallel with our work, Szirmay-Kalos et al. [**?**] conducted research on approximating ray-geometry intersection estimation using distant imposters which they applied to rendering caustics. Although this technique is similar to ours, there are two main differences. We present an intersection estimation algorithm which maps to the Newton-Raphson root finding method that has faster convergence than the Secant method used by Szirmay-Kalos et al. [**?**]. Furthermore, a ray-plane intersection operation is performed for every iteration of their method which is not needed in our algorithm.

5.3 Rendering Caustics

Our rendering algorithm consists of two main phases: (i) construction of a caustic map texture, and (ii) application of the caustics map to diffuse surfaces called receivers. We will first give a brief explanation of how caustics are formed, and then relate it to our method of construction of the caustic-map.

5.3.1 Caustic formation

Caustics are formed when multiple rays of light converge at a single point. This occurs in the presence of refractive or reflective objects which cause the light rays to deviate from their initial path of propagation and converge at a common region. Therefore, to obtain caustics accurately, one must trace light rays from their source and follow their paths through refractive and off reflective surfaces. The photons eventually get deposited on nearby diffuse surfaces, called receivers, thus forming caustics as illustrated in Figure 5.3. Our algorithm closely emulates this physical behavior, and is capable of obtaining both refractive and reflective caustics. For reflective caustics we support a single specular bounce. In the case of refractive caustics, in addition to single surface refraction, our method also supports double surface refraction employing the image-space technique recently proposed by Wyman [Wym05]. The algorithm supports point and directional lighting. Area lights can be accommodated by sampling a number of point lights.

5.3.2 Caustics Mapping Algorithm

Without loss of generality, we will describe rendering of caustics through refractive objects using our algorithm. A general overview of the algorithm, which closely resembles shadow mapping, is presented next along with elaborative discussion on certain parts.

Following is a stepwise breakdown of the main caustics mapping algorithm. Since the algorithm runs entirely on the GPU, it is explained in terms of render passes performed on the graphics hardware.

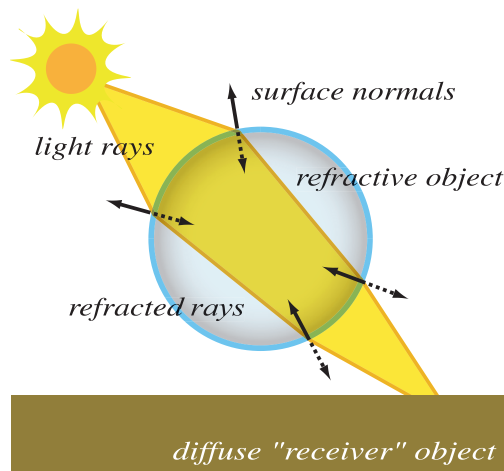


Figure 5.3: Diagram showing how multiple light rays can refract through an object and converge at the same point on a diffuse surface.

- **Obtain 3D positions of the receiver geometry:** The receiver geometry is rendered to a *positions texture* from the light's view. 3D world coordinates are output for each pixel instead of color. This positions texture is used for ray-intersection estimation in the next step.
- **Obtain 3D positions and surface normals of the refractive object:** The refractive object is rendered to texture from the light's view. Using multiple render targets, the surface normals and 3D

world coordinates are output for each pixel. These textures are used with a grid of vertices of equal resolution, such that each vertex maps to a pixel on the texture. The vertex grid is used for the remainder of the algorithm in place of the refractive object.

- **Create caustic-map texture:** The caustic-map texture is created by splatting points onto the receiver geometry from each vertex of the refractive vertex grid along the refracted light direction. The intersection point of the refracted ray and the receiver geometry is estimated using the positions texture. This step is explained in detail in the following section.

- **Construct shadow map:** Although optional, conventional shadow mapping can easily be integrated into the caustics mapping algorithm to render images with both caustics and shadows.

- **Render final scene with caustics:** The 3D scene is rendered to the frame buffer from the camera's view. Each point of the receiver surface is projected into the light's view to compute texture coordinates for indexing the caustic-map texture. The caustic color from the texture is assigned to the pixel and augmented with diffuse shading and shadowing.

The steps mentioned above are performed every frame in separate render passes, and thus impose no constraints on the dynamics of the scene. Furthermore, no computation is performed on the CPU; neither as a per-frame operation nor as a pre-computation.

5.3.3 Creating the Caustic-Map

Rendering of the caustics map texture lies at the heart of our algorithm. It consists of three main steps:

- Refraction of light at each vertex of the refracting surface
- Estimation of the intersection point of the refracted ray with the receiver geometry
- Estimation of caustic intensity at the intersection point

The caustic-map texture is created by rendering the refractive vertex grid from the light's view. The vertices are displaced along the refracted light direction by the estimated distance and then splatted onto the receiver geometry by rendering point primitives instead of triangles. The algorithm is implemented in a vertex shader program and proceeds as follows:

for each *vertex v* **do**

$\vec{r}$ = *Refract(LightDirectionVector)*

P = *EstimateIntersection(v.position,* $\vec{r}$,

ReceiverGeometry)

v.position = P

return *v*

Notice that multiple vertices can end up at the same position on the receiver geometry. In the pixel shader, the light intensity contribution from each vertex is accumulated using additive alpha blending. Details regarding the intensity and final caustic color computation are given in Section 5.3.5.

The intersection point of the refracted ray with the receiver geometry must be computed in order to output the final position of the vertex being processed. Normally, this requires expensive ray-geometry intersection testing which is not feasible for real-time applications. We present a novel image-space algorithm for estimating the intersection point which, to our knowledge, has not been used before.

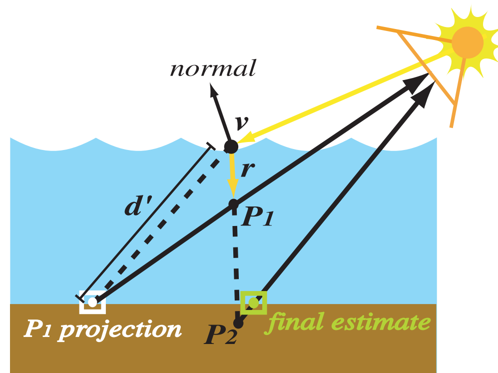


Figure 5.4: Diagram of the intersection estimation algorithm. The solid-lined arrows correspond to the position texture lookups.

The intersection estimation algorithm utilizes the positions texture rendered in the first pass of the main caustics mapping algorithm. A schematic illustration of the procedure is shown in Figure 5.4. Let v be the position of the current vertex and $\overrightarrow{r}$ the normalized refracted light vector. Points along the refracted ray are thus defined as:

$$P = v + d * \overrightarrow{r} \tag{5.1}$$

where d is the distance from the vertex v. Estimating the point of intersection amounts to estimating the value of d, the distance between v and the receiver geometry along $\overrightarrow{r}$. An initial value of 1 is assigned to d and a new position, P_1, is computed:

$$P_1 = v + 1 * \overrightarrow{r} \tag{5.2}$$

P_1 is then projected into the light's view space and used to look up the positions texture. The distance, *d'*, between v and the looked up position is used as an estimate value for d in Equation 5.1 to obtain a new point, P_2. Finally, P_2 is projected in to the light's view space and the positions texture is looked up once more to obtain the estimated intersection point. Following is a snippet of shader language code that computes the estimated intersection point. The inputs to this algorithm are the initial point on the ray, v, the normalized refraction direction vector, $\overrightarrow{r}$, and a texture sampler, *posTexture*, used to lookup the positions texture.

```
float3 EstimateIntersection ( float3 v, float3 r,
        sampler posTexture ) {

float3 P1 =  v + 1.0*r ;
float4 texPt =  mul( float4( P1, 1 ) , mViewProj );
float2 tc = float2(0.5*(texPt.xy/texPt.w)+float2(0.5,0.5));
tc.y = 1.0f - tc.y;
float4 recPos = tex2D(posTexture, tc);
float4 P2 =  v +distance(v,recPos.xyz)*r;
```

```
texPt = mul( float4( P2, 1 ) , mViewProj );

tc = float2(0.5*(texPt.xy/texPt.w)+float2(0.5,0.5));

tc.y = 1.0f - tc.y;

return recPos = tex2D(posTexture, tc);

}
```

5.3.4 Convergence of Intersection Estimation Algorithm

The intersection estimation algorithm is essentially an iterative process of which a single iteration has been discussed above. For example, in the next iteration the distance between the estimated intersection point in the last iteration and *v* would be plugged into Equation 5.1 as a new estimate value for *d*. This iterative process was derived from the Newton-Raphson root finding method, whose convergence rate is faster than other popular methods such as the Secant and Bisection methods. In this sub-section, we map the problem of intersection estimation to a root finding problem. We then discuss the Newton-Raphson method and relate it to our intersection estimation algorithm.

In order to compute the intersection using a root finding method, we must define a function, $f(x)$, whose zero corresponds to the point of intersection of the refracted ray and the receiver geometry. Let the refracted ray be the x-axis for a discrete function, $f(x)$. $f(x)$ is the distance between x and the projection of x. For each

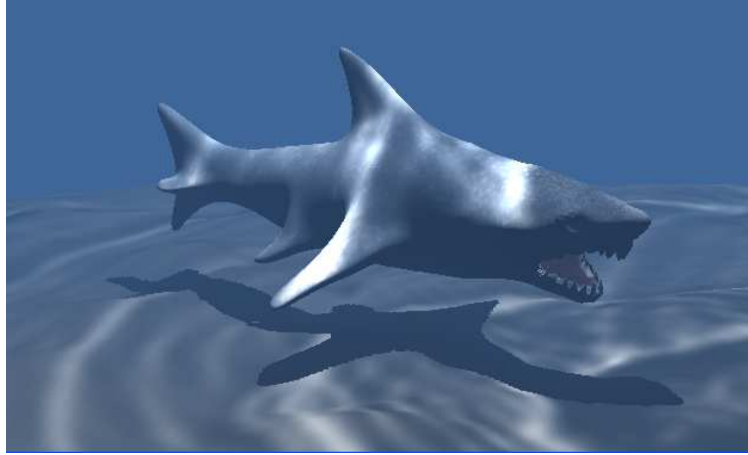


Figure 5.5: Caustics on non-planar surfaces can be obtained using Caustics Mapping. The scene shows underwater caustics on a shark and a rugged seabed.

point, x_i, along the refracted ray at a distance d_i from the origin of the ray, *f(x)* is defined as follows:

$$
\begin{aligned}
y_i &= posTexture(x_i) \\
f(x_i) &= distance(x_i, y_i)
\end{aligned}
\tag{5.3}
$$

The function *posTexture*(x_i) retrieves a 3D point from the *positions texture* by projecting x_i into the light's view and computing texture coordinates. For example, in Figure 5.4, the distance between P_1 and P_1 *projection* is the value for $f(P_1)$.

For simplicity, *f(x)* can be re-parameterized using the distance, *d*, from the origin of the ray instead of a 3D point, *x*, along the ray: $f(v + d * \overrightarrow{r})$ or simply, $\hat{f}(d)$ (recall that *v* and $\overrightarrow{r}$ are the origin and the direction of the refracted ray). Notice that when *d* is equal to the distance from the refracted ray origin to the intersection point, $\hat{f}(d)$ is equal to 0. Therefore, the root of $\hat{f}(d)$ directly corresponds to the distance to the intersection point. We can now define the following theorem:

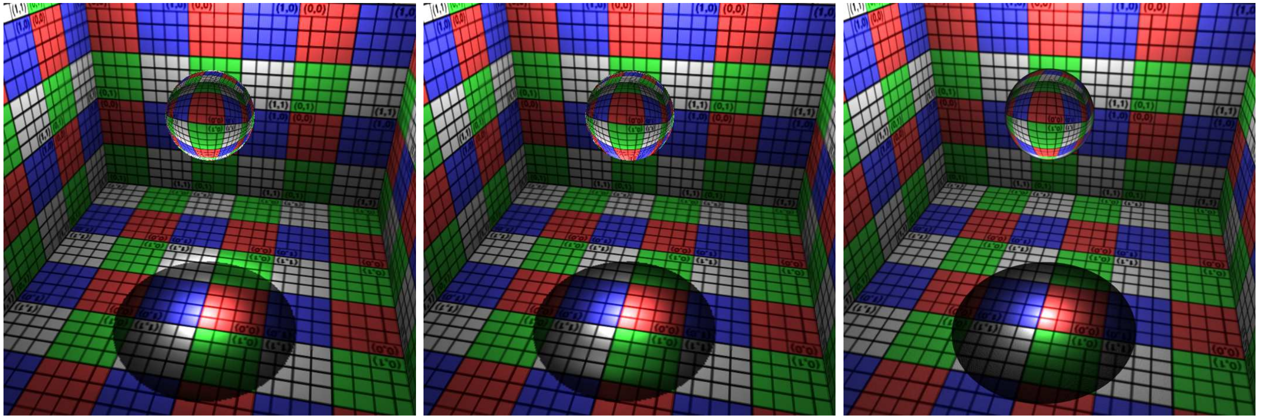


Figure 5.6: Caustics from a glass sphere (from left to right) using the distance imposters method, our caustics mapping, and photon mapping. 3 iterations of the respective intersection estimation algorithms were used for both the first and the second image. The distance imposter's method produced a frame rate of 160 fps whereas caustics mapping produced 245 fps on a GeForce 7800.

Theorem 1. *$\hat{f}(d) = 0$ if and only if* d *is equal to the distance along the refracted ray from the ray origin to the intersection point.*

Proof. We will construct a proof by applying the definition of $\hat{f}$. Let d be the distance along the refracted ray from the ray origin to the intersection point. Let $p = v + d * \overrightarrow{r}$, where v is the ray origin and $\overrightarrow{r}$ is the ray direction. If the *positions texture* is looked up by projecting p into the light's view, the same point p is obtained. Since the distance of a point from itself is 0, the value of $\hat{f}$ must be equal to 0 by definition. $\square$

Thus the intersection estimation problem has been reduced to the problem of computing the root of $\hat{f}$. Although any root finding algorithm can be applied as per user preference, the Caustics Mapping algorithm uses an iterative method derived from the Newton-Raphson (NR) algorithm. The NR iteration is defined as follows:

To find a root of $\hat{f}(d) = 0$ with the initial guess d_0,

$$d_{k+1} = d_k - \frac{\hat{f}(d_k)}{\hat{f}'(d_k)} \text{ where } k = 0, 1, 2, 3, ... \tag{5.4}$$

Note that the NR iteration requires the evaluation of the derivative function, $\hat{f}'(d)$. Since the function $\hat{f}$ is in a discrete form, its derivative can be computed using finite differences.

$$\hat{f}'(d) = (\hat{f}(d + \Delta) - \hat{f}(d))/\Delta \tag{5.5}$$

GPU code for the NR iteration is provided in the Appendix, and it can be used in place of our approximative iterative solution if desired with the added cost of computing the derivative. In certain cases where the derivative is approximated using the difference between two points specifying an interval, this iteration is referred to as the Secant method. Note that the Secant method operates on an interval of arbitrary length, whereas to compute the derivative of a function using finite difference in the NR method, the interval has to be small. Computing the derivative requires two evaluations of the function $\hat{f}$. This directly translates to two accesses of the *positions texture* per iteration. In order to reduce the overall number of texture accesses, we introduce an approximation to the NR evaluation. The NR method gives d_{k+1}, the new estimate of the root point by incrementing d_k, with a scaled version of $\hat{f}(d_k)$, the function value at d_k. The exact scale is the

negative reciprocal of the derivative of the function at d_k. In our method we use the distance of the surface geometry at d_k from the origin of the refracted ray (i.e. d_0=0) as the estimate for d_{k+1}. This approximation of the NR iteration was influenced by three key observations:

- **A rough similarity to NR estimator:** The distance of the surface geometry at d_k from the origin of the refracted ray (i.e. d_0=0), can be approximated as d_k incremented by an amount related to $\hat{f}(d_k)$.
- **Correctness:** At the root, i.e. at the point of intersection, d_k is indeed equal to the distance of the surface geometry from the origin of the ray.
- **Computational efficiency:** This estimator eliminates the need for explicitly computing the function derivative and thus cuts down the number of texture lookups by half.

The intersection estimation algorithm assumes that the true intersection point lies at the same distance from v as the current estimate point. Visually, this can be observed as a circular arc connecting P_1 and P_2, centered at v with radius *d'*. The assumption that the receiver surface is curved rather than planar in the local region of the intersection was motivated by the fact that no explicit intersection tests would be required in the former case. The algorithm thus incurs the cost of just a texture look-up at each iteration, unlike the distant imposters technique which additionally requires a ray-plane intersection test.

As with all iterative root finding methods, the continuity of the function is a vital criterion for convergence. Thus in order for the intersection estimation to be successful, the visibility of the receiver geometry from the light's view must be well defined. Note that when the *positions texture* lookup returns a null value, it implies that no receiver geometry was seen in that direction. If this null value is encountered in the intersection estimation algorithm, the result obtained is incorrect. In practice, the value returned is a point at infinity.

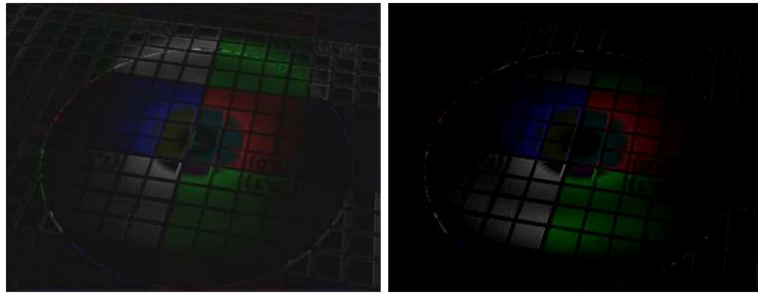


Figure 5.7: Difference images of (Left) the distance imposters method, and (Right) our caustics mapping method with photon mapping. The difference images were obtained from the results shown in Figure 6.

Such a situation arises near the edges of the reciever geometry. However, this only means that some of the caustic splats would be lost, but it does not cause any visual artifacts in the final caustics rendering. Another limitation that is specific to the Newton-Raphson method is that in some cases it starts oscillating between two reference points and thus never converges to the true root. This problem is attributed to the fact that the Newton-Raphson method keeps only the most recent estimated point instead of an interval which contains the root point. This oscillation, however, can be detected by storing the last two estimates and then remedied by switching to a more conservative algorithm such as the Bisection method.

We have found empirically that the estimate of the intersection point improves rapidly with each iteration. The magnitude of the error and the number of iterations to convergence depends on the topology distribution of the scene. For a fairly uniform topology, the error is negligible after only 5 iterations. For the purpose of rendering caustics, we observed that only a single iteration is sufficient. Furthermore, the small error in the estimation is well sustained by our caustics mapping algorithm and thus is suitable for the purpose. It is important to note that the algorithm is not limited to rendering caustics on planar surfaces. As long as the visibility function is well defined and continuous, even fairly rugged terrains produce accurate results.

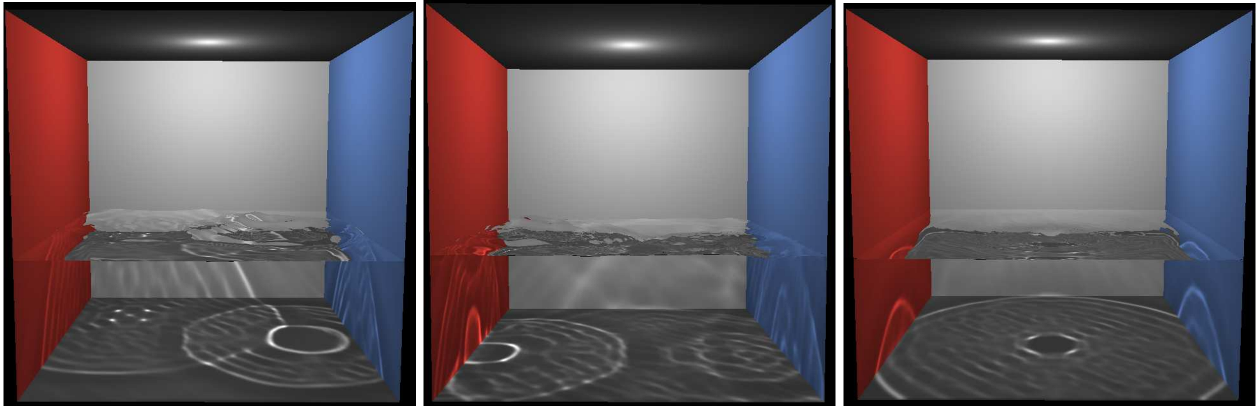


Figure 5.8: Frames from a water animation demo with caustics mapping. Random ”plops” are introduced in the water surface mesh at regular time intervals. Notice the caustics pattern and the corresponding shape of the water surface.

Figure 5.5 shows under-water caustics falling on a shark and the floor, both of which have non-planar surface geometry.

5.3.5 Caustic Intensities

The intensity of the caustics formed on the diffuse receiver geometry depends on the amount of light that gets accumulated at any particular point on the receiver's surface. Since the caustics map texture is created by refracting or reflecting the light rays at each vertex, the intensity of the caustics will depend on the number of vertices which make up the caustic-forming object. If the algorithm is employed using a naive accumulation scheme, the caustics will appear to be bright or dark depending on the number of vertices of the refractive object. This undesirable phenomenon occurs due to the incorrect assumption that the light flux contribution for each vertex is the same. We combat this problem by computing a weighting coefficient for each vertex which is then multiplied with the incident light.

In order to compute the coefficient for a vertex, the total flux contribution for that vertex must be determined by observing the total flux through the surface of the caustic-forming object visible to the light souce. Therefore, the total flux contribution for each vertex in the grid is computed as:

$$\Phi_i = \frac{1}{V}(N_i \cdot L_i) \tag{5.6}$$

where V is the total projected visible surface area of the object, N_i is the surface normal vector at the vertex, and L_i is the incident light vector. V is determined by counting the number of vertices of the refractive grid that are occupied by the projection of the refractive object. Since the grid resolution is the same as that of the render target texture(s) used in Step 2 of the Caustics Mapping algorithm, the number of rasterized pixels in that step gives the value for V. The process of determining the number of rasterized pixels is known as occlusion querying, and this feature is generally implemented in modern graphics APIs. The occlusion query simply returns the number of pixels that pass the Z-test.

It is important to note that the caustic intensity computation is performed every frame since the projected visible surface area can change. However, it does not involve any complex operations that can have a significant impact on the frame-rate.

The final color of the caustics formed through a refractive object is computed by taking into account the absorption coefficient of the object's material. This enables us to achieve colored caustics as shown in Figure 5.1 and Figure 5.13.

$$I = I_o e^{-K_a d} \tag{5.7}$$

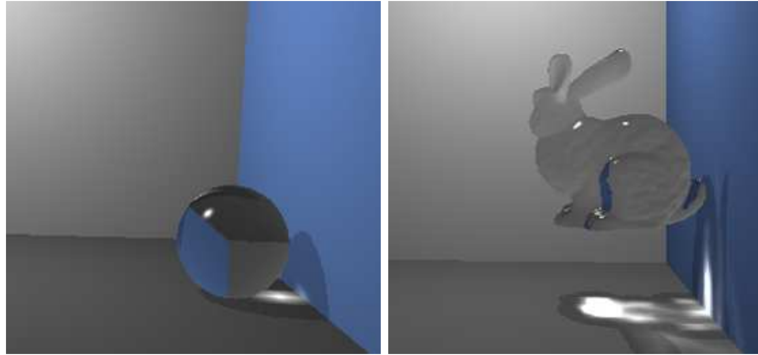


Figure 5.9: Caustics rendered using our algorithm for simple objects such us spheres (245fps) as well as complex ones such as the Stanford bunny (200fps)

where I_o is the incident light intensity, K_a is the absorption coefficient, and d is the distance that light travels through the refractive object. This distance is easily determined by rasterizing the back-faces of the object in an initial render pass to obtain positions of the hind points. In fact, this data is already available if double surface refraction is used [Wym05]. For single refractive surfaces such as water, attenuation is performed over the distance between the surface and the receiver geometry.

5.3.6 Implementation Issues

The caustics mapping algorithm suffers from issues similar to shadow mapping and other image-space techniques. There are two main points of concern that must be addressed: aliasing, and view frustum limitation. The former issue is inherent in all image-space algorithms. This problem is further magnified due to the usage of point primitives rather than triangles for rendering of the caustic-map. The gaps between the point splats give a non-continous appearance to the caustics. However, if there are sufficient number of vertices in the refractive vertex grid, the gaps are significantly reduced. Figure 5.11 shows a comparison

of the caustics produced with two different refractive grid resolutions: 64×64 and 128×128. In our implementations, we have used grids of 128×128 vertices. It is important to note that the tessellation of the mesh of refractive object has no effect on the quality of the caustics, which is solely dependent on the resolution of the refractive grid. Further antialiasing can be achieved by using textured point primitives with a gaussian fall-off for splatting. This produces continuity between the individual point splats, and was employed for generating all the images in this paper. Although not required, the caustic-map texture can also be low-pass filtered for additional smoothening of the caustics.

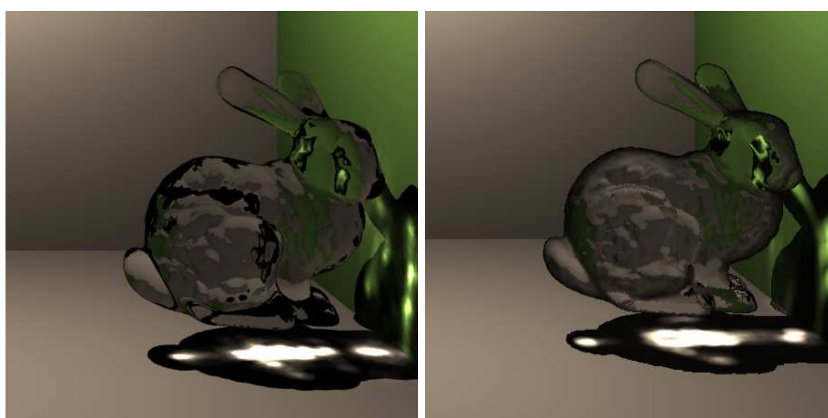


Figure 5.10: Comparison of photon mapping (Left) and Caustics Mapping (Right) for a complex refractive object.

The second issue pertaining to the algorithm is the view frustum limitation during rasterization of the caustic-map. This problem is exactly like that of shadow mapping with point lights. In such a case an environment shadow map is created, usually with six 2D textures corresponding to each face of a cube map rather than a single texture. Similarly, in the caustics mapping algorithm if the caustics are formed outside the light's view frustum, they will not be captured on the caustic-map texture. This happens more frequently with reflective caustics than refractive ones. Using an environment caustic map solves this problem at an overhead cost of rendering extra textures. In our implementation we employed cube maps, however the dual

paraboloid mapping technique proposed by Heidrich [Hei98], which has been applied to shadow mapping for omnidirectional light sources by Brabec et al. [BAS02], can also be utilized.

5.4 Results and Limitations

The caustics mapping algorithm was developed and implemented on a 2.4GHz Intel Xeon PC with 512MB of physical memory. However, since the algorithm performs no computation on the CPU, a lower-end processor will be sufficient. We employed a GeForce 7800 graphics card and pixel shader model 3.0 because the algorithm requires texture access in the vertex shader for intersection estimation in the caustic-map generation step. The algorithm was implemented using Microsoft DirectX 9.0 SDK.

The major advantage of our algorithm is the speed at which it renders the caustics, making it very practical for utilization in games and other real-time applications. We conducted a number of tests and produced results to demonstrate the feature-set of our algorithm. Caustics from both refractive and reflective objects were generated. For refractive objects, two main categories were established: single-surface refraction, and double-surface refraction. Single-surface refraction is suitable for underwater caustics since there is only a single refraction event as light enters through the water surface and hits the floor. The result of our underwater caustics can be seen in Figure 5.8. It was rendered at a resolution of 640×480 at the rate of 60 frames per second. The mesh used for the water surface consisted of 100×100 vertices.

For solid refractive objects, using double-surface refraction is more physically correct. Most of the previous work done in real-time caustics rendering is limited to single-surface refraction and cannot be easily

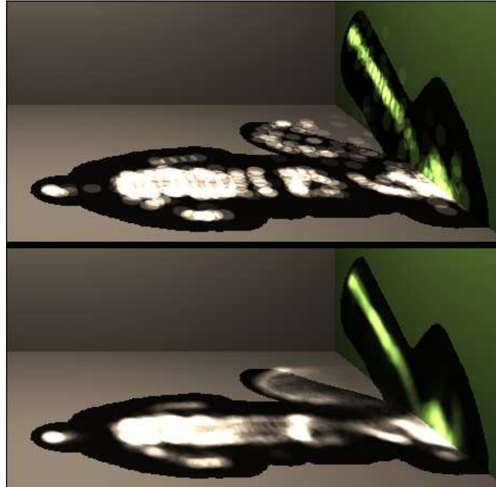


Figure 5.11: Effect of refractive grid resolution. The top image was rendered using a grid resolution of 64×64, whereas the bottom image was rendered using a grid resolution of 128×128. The gaps between point splats that are visible at grid resolution 64x64, are not noticeable at grid resolution 128x128

extended to include the double-surface interaction. Our algorithm effortlessly accommodates Wyman's [Wym05] image-space double-surface refraction. We rendered various simple and complex objects and observed the caustics that they produced. The results with caustics from a sphere, and the Stanford bunny are shown in Figure 5.9. Notice that the frame-rate even with double-surface refraction is quite high. The shadows in these images were obtained using conventional shadow mapping. Since we deal with point lights, a shadow cube map was employed. The resolution of both the shadow map and the caustics map was $768 \times 768 \times 6$ pixels for rendering final images of resolution 1024×768 pixels. Figure 5.1 shows caustics from the Stanford bunny up close, occupying most of the caustic map region. The frame-rate in this case is 42fps, which shows that the algorithm is fill-rate dependent.

Figure 5.10 shows a comparison of images rendered with Caustics Mapping and an offline rendering system using photon mapping. Notice that even with complex objects, such as the Stanford bunny, our algorithm produces visually convincing results. Some small differences do exist, and they are mainly attributed to the

accuracy of the emulation of the refraction events that take place as light travels through the bunny to the receiver surface. For simpler geometry, such as spheres, the caustics produced using our algorithm are closer to those produced with photon mapping since the refraction events are fairly simple and can be accurately emulated using double surface refraction. Figure 5.6 shows a comparison of the distant imposters method, our caustics mapping algorithm, and photon mapping using PBRT. The difference images are shown in Figure 5.7. The difference image on the right shows that the caustics generated by our method are similar to the caustics generated from classical photon mapping algorithm.

Caustics from reflective objects using our method are shown in Figure 5.12. Our caustics mapping algorithm performs better with refractive caustics than reflective ones since the error in the intersection estimation during the caustic-map generation step tends to be greater in the latter case. This is due to the fact that refraction causes small deviations in the path of the light ray, whereas reflection causes the ray to change its direction completely. Therefore for the reflection case, the visibility function is more probable to be discontinuous. We would like to improve upon our intersection estimation algorithm in future work to better handle this issue.

Another limitation of our algorithm is that it does not easily extend to volumetric caustics like some techniques proposed in previous work [EAJ05]. This is mainly due to the fact that the caustic-mapping algorithm operates in image-space. One way of achieving volumetric caustics using our algorithm is to use a number of planes perpendicular to the light ray as the caustic receivers. However this method spawns further issues relating to sampling and volumetric rendering which need to be addressed.

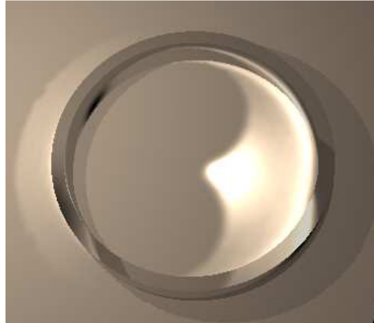


Figure 5.12: Reflective caustics from a metal ring. The desired shape that is observed in real life is obtained from the caustics-mapping algorithm. Notice the circular caustic band outside the left edge of the ring.

Figure 5.13: (Left) Colored caustics from a refractive bunny with light attenuation in different color channels. (Right) Spectral refraction effect obtained using the caustic mapping algorithm with different refractive indices for each of the three RGB color channels.

5.5 Conclusion and Future Work

We have presented a practical real-time caustics rendering algorithm that runs entirely on the graphics hardware and requires no pre-computation. It emulates the light transport involved in caustics formation in the image-space; therefore it is both physically inspired and fast. The algorithm is conceptually similar to shadow mapping and integrates easily into virtually any rendering system.

We also presented a fast ray-scene intersection estimation technique which we plan to further explore and improve in future work. This technique is extremely fast and is well suited to algorithms, such as caustics mapping, which can tolerate a certain amount of error in the estimation.

Finally we would like to develop methods of applying the caustics mapping algorithm to participating media and achieve volumetric caustics.

Acknowledgment

The authors would like to thank Guillaume Francois, Kévin Boulanger, and Jared Johnson for their insightful comments, and Eric Risser for assistance with illustrations used in this paper. This work was supported in part by I2Lab, Electronic Arts and ATI Reseach.

Appendix

The following code listing is a GPU implementation of the Newton-Raphson root finding method for the intersection estimation computation. Note that this is the original, unmodified version of the Newton-Raphson algorithm. In our work, we use a modified version which does not require the computation of the derivative and is thus cheaper to evaluate. The input parameters to the function are respectively: *pos*, the origin of the refracted ray; *dir*, the direction of the refracted ray; *mVP*, the light's View-Projection matrix; *posTex*, the

receiver geometry positions texture; *num_iter*, the desired number of iterations. The function *getTC()* used in the code computes texture coordinates from a 3D point by projecting it to the texture's view plane and moving it to the range [0,1], identical to computing shadow mapping coordinates. The function returns the distance along the refracted ray as an estimate of the intersection point.

```
float rayGeoNP(float3 pos, float3 dir, float4x4 mVP,
               sampler posTex, int num_iter)
{
    float eps = 0.1;
    float2 tc = float2(0.0,0.0);

    // initial guess
    float x_k = 0.10;
    for(int i=0;i<num_iter;i++)
    {
        // f(x_k)
        float3 pos_p = pos + dir*x_k;
        tc = getTC(mVP, pos_p);
        float3 newPos1 = tex2D(posTex, tc);
        float f_x_k = distance(newPos1 , pos_p);

        // f(x_k + eps)
        float3 pos_q = pos + dir*(x_k+eps);
        tc = getTC(mVP, pos_q);
        float3 newPos2 = tex2D(posTex, tc);
        float f_x_k_eps = distance(newPos2 , pos_q);

        float deriv = (f_x_k_eps - f_x_k)/eps;

        // the newton raphson iteration
```

```
        x_k = x_k - (f_x_k/deriv);
    }

    return x_k;
}
```

CHAPTER 6

IMAGE-SPACE SUBSURFACE SCATTERING FOR INTERACTIVE RENDERING OF DEFORMABLE TRANSLUCENT OBJECTS

6.1 Introduction

Most real world materials exhibit translucency to some extent at an appropriate scale. The subsurface interaction of light involves the light entering through the surface and scattering multiple times inside the material, ultimately exiting the surface in a diffuse manner. For materials with relatively low absorption, but high scattering property, such as marble, the light scatters around a larger distance under the surface and hence exits the surface at locations other than where it entered. Extent of the translucency depends on the scattering and absorption property of the material. Certain materials that are highly scattering give rise to unique optical phenomena, such as the appearance of a backlit object (example: Figure 6.7), that distinguish their visual look from other opaque solids. An important goal in computer graphics is to faithfully represent these materials for realistic image synthesis. BRDFs (Bidirectional Reflectance Distribution Functions) are

Figure 6.1: From left to right: bird model rendered (a) using Lambertian diffuse and Phong specular lighting, (b) our subsurface scattering algorithm (27 fps), and (c) an offline renderer (approx. 3 mins per frame). Note the translucency at the cheek and feet regions in the latter images.

generally used to model the reflectance properties of most opaque materials. This modeling is based on the assumption that light enters and exits from the same location on the surface. Because of this assumption, BRDFs are not sufficient for translucent objects. Instead, translucency is modeled using the BSSRDF (Bidirectional Surface Scattering Reflectance Distribution Function), which takes into account both the angular and spatial distribution of light on the surface of an object. A BSSRDF is an 8-dimensional function that describes the light transport from one point to another for a given illumination and viewing direction.

A number of offline rendering methods that perform numerical simulation of subsurface scattering have been used for rendering of translucent materials. Although the results produced by these methods are highly convincing, the computation involved is generally very expensive. With the recent introduction of the dipole BSSRDF model by Jensen et al. [JML01] that employs the dipole source approximation for multiple scattering, a number of fast rendering algorithms have emerged. Our work, inspired by the dipole BSSRDF model, is focused on developing a real-time algorithm for rendering translucent materials with no limitations on lighting, viewing, and object deformation, while retaining the visual quality comparable to offline rendering results (Figure 6.1). Our main contribution is that we use a dual light-camera space technique to efficiently

evaluate the subsurface scattering by only considering interaction between significant point pairs. The area integral is computed via a “splatting approach” in an image-space framework, thus allowing support for arbitrarily complex geometry. We show that our algorithm can be implemented entirely on the GPU and is easily integrated into existing real-time rendering systems by inserting only three additional render passes.

6.2 Previous Work

Subsurface scattering in translucent materials has been extensively studied in computer graphics. Impressive results have been produced from offline rendering techniques that simulate light transport in participating media. Although these approaches are able to faithfully reproduce most of the subsurface scattering effects, they are generally slow. Faster algorithms have emerged after the introduction of the work by Jensen et al., which employs the dipole source approximation for multiple scattering in the BSSRDF model [JML01]. A two-pass hierarchical approach that significantly increased the speed of computation of the multiple scattering term was presented soon after [JB02]. In this approach, irradiance samples are taken on the surface of the object which are then organized into an octree data structure. This allows for hierarchical integration where the neighboring points are sampled densely and those further away are considered in groups. Recent methods have built up on this technique and accelerated the multiple scattering computation further using graphics hardware. These algorithms closely resemble radiosity; the scattering event is treated as point-to-point [LGB02, HV04], point-to-patch [MKB03b], and patch-to-patch [CHH03] form-factor like transport. Although rendering at interactive frame-rates has been shown using these techniques, they are unable to efficiently handle environment lighting. Furthermore in order to maintain interactivity, pre-computation is

required for establishing the form-factor links in most cases. Unlike its counterparts, the technique presented by Mertens et al. [MKB03b] is able to handle deformable geometry. They provide three different rendering modes of which the "on-the-fly" mode offers the most flexibility since it involves full evaluation from scratch. However it is also the slowest rendering mode producing an average frame-rate of about 9fps.

Subsurface scattering was included in the precomputed radiance transfer (PRT) framework by Sloan et al. in [SHH03]. They precompute and store the multiple scattering component of the BSSRDF as transport vectors in truncated lower order spherical harmonics (SH) basis. Scattering SH coefficients are then combined with the SH coefficients of the environment lighting function at render stage. Support for deformable objects was added to the PRT framework in [SLS05]. Support for all-frequency lighting was first introduced for the case of diffuse BRDF rendering by Ng et al. [NRH03] and was recently adapted by Wang et al. [WTL05] to include full BSSRDF rendering accounting for both the single and multiple scattering terms.

The goal of our work is to provide a GPU-based real-time algorithm for rendering deformable translucent objects with dynamic lighting and viewing. Deviating from traditional convention of operating on geometry, we adopt an image-space approach, which allows us to efficiently employ the graphics hardware for the scattering computation as well as support objects of arbitrary geometric complexity without compromising real-time framerates.

Image-space algorithms operate on rasterized images of 3D scenes rather than the geometry, and hence are very suitable for implementation on GPU. Recent advances in graphics hardware have enabled researchers to develop fast image-space algorithms for rendering complex optical effects such as reflection, refraction, and caustics [SAL05, SKP07a, Wym05, WD06]. Steps towards real-time global illumination are also being taken in the same spirit. Dachsbacher and Stamminger show real-time indirect illumination for diffuse and

Table 6.1: Symbols used in the multiple scattering computation using the dipole source BSSRDF model.

Symbol	Description
$\overrightarrow{\omega_i}$	incident light direction
$\overrightarrow{\omega_o}$	exiting light direction
n_i	surface normal
g	average cosine of scattering angle
σ_s	scattering coefficient
σ_a	absorption coefficient
σ_s'	reduced scattering coefficient = $(1-g)\sigma_s$
σ_t	extinction coefficient = σ_a + σ_s
σ_t'	reduced extinction coefficient = σ_a + σ_s'
η	relative refraction index
$F_t(\eta, \overrightarrow{\omega})$	Fresnel transmittance
$p(\overrightarrow{\omega_i}, \overrightarrow{\omega_o})$	phase function

non-diffuse surfaces using an image-space splatting approach [DS06]. Mertens et al. [MKB03a] present an image-space subsurface scattering algorithm that utilizes the dipole source approximation. They precompute a set of sample points for the area integration and evaluate the integral using multiple GPU passes. Similarly, Dachsbacher and Stamminger present an algorithm that employs translucent shadow maps [DS03], treating the integration computation as a filtering operation. Since [MKB03a] and [DS03] most closely relate to our work, we provide a short comparative discussion of these techniques after explaining our algorithm in Section 6.4.4.

In this paper, we present an image-space algorithm for real-time subsurface scattering in translucent materials. Our method also utilizes Jensen's dipole source approximation BSSRDF model and supports both point light and environment illumination. The algorithm does not require any precomputation and is evaluated fully at every frame. We obtain results comparable to those produced using offline renderers at about 25 frames per second for arbitrarily complex objects.

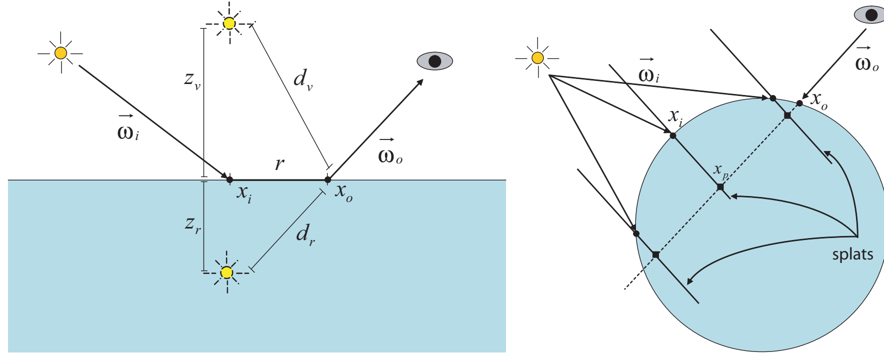


Figure 6.2: (a) Dipole source approximation for multiple scattering. (b) Diagram of our rendering algorithm. The points x_p on the individual splats project on to the visible surface points x_o. The multiple scattering term is evaluated at each x_p by taking the distance between x_o and the corresponding x_i. The results are accumulated using additive alpha blending and stored at x_o.

6.3 Background

Our algorithm employs the BSSRDF model for computing the subsurface scattering in a given homogenous object. BSSRDFs are one level higher up than BRDFs in the hierarchy of models for describing object appearance. A BRDF is a 4-dimensional function that describes the angular distribution of the outgoing radiance relative to the incoming radiance at a single point. However, translucency cannot be faithfully accounted for with such a representation. This is because translucent materials exhibit subsurface scattering, in which light entering at a certain point gets transmitted inside the medium and can exit at a different location. The BSSRDF is an 8-dimensional function that additionally describes the spatial distribution of light between two points on the surface. Therefore the exiting radiance at point x_o in direction ω_o is given by:

$$dL_o(x_o, \overrightarrow{\omega_o}) = S(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o}) d\Phi(x_i, \overrightarrow{\omega_i}) \tag{6.1}$$

$$L_o(x_o, \overrightarrow{\omega_o}) = \int_A \int_{2\pi} S(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o}) L_i(x_i, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) d\omega_i dA(x_i) \tag{6.2}$$

The various symbols used throughout this paper are defined in Table 6.1. For highly scattering, optically thick materials, the BSSRDF, $S(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o})$, is approximated as a sum of a single scattering term $S^{(1)}$ and a diffuse multiple scattering term S_d:

$$S(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o}) = S^{(1)}(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o}) + S_d(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o}) \tag{6.3}$$

It has been shown that for highly scattering translucent materials it is sufficient to only consider the diffuse multiple scattering term of the BSSRDF. Most of the current interactive BSSRDF rendering techniques therefore do not account for the single scattering term which enables them to simplify computation and consequently increase speed.

6.3.1 Multiple Scattering

The multiple scattering term accounts for the diffusion of light inside the material. Although an analytic solution for multiple scattering is not known, it can be estimated using the dipole source approximation defined by Jensen et al. as follows:

$$S_d(x_i, \overrightarrow{\omega_i}, x_o, \overrightarrow{\omega_o}) = \frac{1}{\pi} F_t(\eta, \overrightarrow{\omega_i}) R_d(||x_i - x_o||) F_t(\eta, \overrightarrow{\omega_o}) \tag{6.4}$$

R_d is defined as:

$$R_d(r) = \frac{\alpha'}{4\pi} \left[z_r \left(\sigma_{tr} + \frac{1}{d_r} \right) \frac{e^{-\sigma_{tr} d_r}}{d_r^2} + z_v \left(\sigma_{tr} + \frac{1}{d_v} \right) \frac{e^{-\sigma_{tr} d_v}}{d_v^2} \right] \tag{6.5}$$

where $\alpha' = \sigma_s'/\sigma_t'$ is the reduced albedo obtained from the reduced scattering and extinction coefficients, and $\sigma_{tr} = \sqrt{3\sigma_a\sigma_t'}$ is the effective extinction coefficient. The dipole approximation treats the multiple scattering contribution to a point x_o of an incoming light ray at the point x_i as illumination from a pair of real-virtual light sources placed below and above the surface at distances z_r and z_v respectively (Figure 6.2(a)). d_r and d_v are distances from x_o to the real and virtual dipole lights. These distances are computed as follows:

$$d_r = \sqrt{r^2 + z_r^2} \tag{6.6}$$

$$d_v = \sqrt{r^2 + z_v^2} \tag{6.7}$$

$$r = ||x_o - x_i|| \tag{6.8}$$

$$z_r = \frac{1}{\sigma_t'} \tag{6.9}$$

$$z_v = z_r(1 + \frac{4A}{3}) \tag{6.10}$$

$$A = \frac{(1 + F_{dr})}{(1 - F_{dr})} \tag{6.11}$$

$$F_{dr} = \frac{-1.440}{\eta^2} + \frac{0.710}{\eta} + 0.0668 + 0.0636\eta \tag{6.12}$$

Substituting S_d in Equation 2 we obtain the outgoing radiance equation:

$$L_o(x_o, \overrightarrow{\omega_o}) = \frac{F_t(\eta, \overrightarrow{\omega_o})}{\pi} \int_A R_d(||x_i - x_o||)E(x_i)dA(x_i) \tag{6.13}$$

where E is the transmitted irradiance defined as:

$$E(x_i) = \int_{2\pi} L_i(x_i, \overrightarrow{\omega_i})F_t(\eta, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i})d\omega_i \tag{6.14}$$

For a point light source, the expression $E(x_i)$ is simpler:

$$E(x_i) = \frac{I(\overrightarrow{\omega_i})}{d_{pl}^2} F_t(\eta, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \tag{6.15}$$

where d_{pl} is the distance from the point light source to x_i. Note that R_d is simply a function of the distance between the incoming and outgoing points, x_i and x_o. Therefore, for a given material with known absorption and scattering coefficients, R_d can be computed and stored as a lookup table indexed by distance. However, computing the outgoing radiance at x_o involves evaluating $R_d(||x_o - x_i||)$ for all sample points x_i over the entire surface of the object. Thus the cost of the final rendering is quadratic in the number of sample points (sample points seen by the light $\times$ sample points seen by the camera), which can be quite expensive. Most recent work has focused on finding ways for performing this integration process efficiently by using hierarchical data structures, pre-computation, etc.

6.4 Our Algorithm

We compute the area integration in Equation 13 by a "splatting" process rather than a "gathering" process. We take sample points on the surface visible from the light source and "splat" the scattering contribution to all the points visible to the viewer within the effective scattering range from each point. Each point on the surface that is rendered receives the scattering contribution from all the points that influence it.

We first describe our rendering algorithm for a single object under point light illumination. Environment lighting is also supported and is described later in Section 6.4.2. Furthermore, since our algorithm operates in

image-space, in Section 6.4.3 we show that multiple objects with different scattering properties are rendered simultaneously without any complication or any additional cost.

Our algorithm consists of the following steps:

Compute dipole approximation lookup table: Equation 5 is computed and stored in a texture for a number of prescribed parameter values similar to existing techniques [JML01]. The range of the distance values used for computing R_d depends on the absorption and scattering coefficients of the material being rendered. Details on computing the maximum distance, r_{max}, are provided in Section 6.4.1. Note that although the dipole equation can be evaluated every frame for every point pair, we choose to store it as a lookup table since it only depends on the material's scattering properties. Hence for a given material, this lookup table is fixed and therefore the computation every frame can be avoided. Furthermore, our system also supports multipole diffusion approximation, in which case the function must be stored as a lookup table since it is too expensive for per-frame evaluation in an interactive application.

Create scatter texture: The scatter texture contains the illumination due to subsurface scattering towards the visible surface of the object. A flowchart of this step is presented in Figure 6.3. The object is first rendered from the light's view in order to obtain sample points x_i on the surface that receive light. Light is then distributed, or "shot", from these points to the points visible to the viewer by rendering screen-aligned quads, also known as "splats", centered at each x_i (Figure 6.2(b)). The size of the splats corresponds to the maximum distance for which the dipole term R_d is computed in the previous step. The quad is projected into the camera's view space to obtain the corresponding visible points x_o on the surface. The dipole approximation lookup table is then queried using the distance between x_o and x_i, and the value is modulated with the transmitted irradiance to obtain the amount of light scattered from x_i to x_o due to

multiple scattering. The final outgoing radiance is obtained by using additive alpha blending to accumulate the scattered light to x_o from all x_i, essentially solving the integral in Equation 13. The alpha channel is incremented by 1 each time the pixel is written to, thus acting as a counter and is used to compute $dA(x_i)$.

Final rendering with subsurface scattering: In this step the scatter texture is simply mapped onto the object and specular highlights are added in.

An animated description of our algorithm is included in the supplementary video for this paper.

It is important to note that the scatter texture is rendered from and stored in the camera's view space. The reason for this dual-space approach is that the light's view captures all the points that receive light. However, light can be scattered to other points on the surface of the object that are not visible from the light's view. Therefore if the scatter texture is stored in the light's view space, those points will not be accounted for and important visual effects such as the scattering in back lit objects will not be attained. On the other hand, if the scatter texture is stored in the camera's view space, scattering to all the visible points will be computed. Although some points that could potentially receive light due to scattering are still left out, they are insignificant since they are not visible to the viewer. Furthermore, other than accurately accounting for the scattering, this dual-space representation also helps in eliminating stretching artifacts at grazing angles due to under sampling that occur when points are projected from camera's view space to the light's view like in shadow mapping. In our algorithm, even though the points x_i are sampled in the light's view, the scattering is performed by rendering splats perpendicularly aligned with the camera's view. In doing so, stretching artifacts are prevented since no conversion from one space to the other is involved.

Our splatting approach has two main advantages over the traditional integration approach of gathering scattering contribution from nearby points. First, we only compute scattering between significant point pairs,

i.e. pairs of splatting and gathering points such that the splatting points must have incident light to scatter, and the receiving points must be visible to the viewer. Secondly, treating the integration computation in this fashion allows for a very simple and natural implementation on the GPU using additive alpha blending.

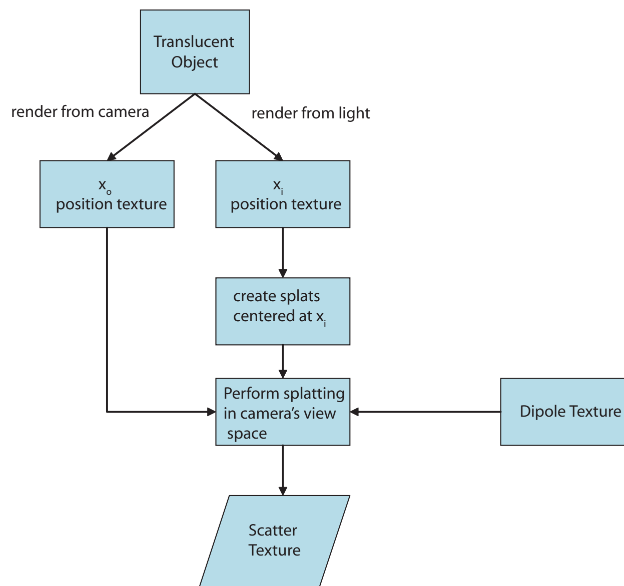


Figure 6.3: Flow chart illustrating the scatter texture creation process.

6.4.1 Algorithm Implementation Steps

Our algorithm runs entirely on the GPU. Therefore it is useful to describe it in terms of render passes. The algorithm consists of three render passes to compute the subsurface scattering texture, and a final render pass in which this texture is mapped onto the object.

Pass 1: Render object from light's view. In the pixel shader, output 3D world space positions and normals to a texture. This gives a set of "splatting" points x_i.

Pass 2: Render object from camera's view. In the pixel shader, output 3D world positions to a texture. This

texture is the set of x_o points.

Pass 3: From the camera's view, render $n \times n$ screen-aligned quads (or splats) to create the scatter texture, where $n \times n$ is the resolution of the texture rendered in Pass 1. Each splat is centered at the corresponding x_i and is parallel to the camera's view plane. Note that the splats are positioned dynamically in the vertex shader. This requires a texture lookup in the vertex shader which is supported by shader model 3.0 and above. Alternatively, the render-to-vertex buffer extension of OpenGL can be used to output splat positions. In the pixel shader, the multiple scattering term at each pixel x_p is computed as explained earlier in the scatter texture creation step. Additive alpha blending is enabled for this pass to accumulate the scattering contribution from each splat.

In the final render pass, the object is rendered with subsurface scattering using the scatter texture and the specular reflections are added in.

Note that there are two variable quantities in the algorithm: the total number of splats used, and the size of individual splats. We first compute the splat size from the the scattering properties of the material being rendered, and then calculate the total number of splats required. The splat size is computed by determining the maximum extent of the spatial domain of integration, r_{max}, such that the integral of the dipole diffusion function over the truncated area $A' = r_{max}^2$ is within some threshold fraction of the integral over the entire surface area, A:

$$\frac{\int_A R_d(x)dA(x) - \int_{A'} R_d(x)dA'(x)}{\int_A R_d(x)dA(x)} < \varepsilon \tag{6.16}$$

The value of r_{max} that satisfies this inequality is computed numerically. r_{max} is used as the splat size, and A' is referred to as the effective scattering range. Next, the dipole function and r_{max} are used to determine the number of overlapping splats, n_o, required at each point in order to compute the integral in Equation 13. We

employ Weber's Law to obtain the number of splats required such that adding more splats will not result in a perceptible change in the final value:

$$\frac{R_d(0)}{n_o \times R_d(r_{max})} < \varepsilon \tag{6.17}$$

n_o can easily be computed from this equation. The error thresholds, ε, used in Equation 16 and Equation 17 can be adjusted by the user to obtain the desired rendering quality and speed. The results reported in this paper have been created using $\varepsilon = 0.01$.

Finally, based on the number of overlapping splats and the size of individual splats, we can compute the total number of splats required in our uniform sampling scheme. If the width of the light's view frustum in world units is W, then the number of splats required is $n \times n$ where:

$$n = \frac{\frac{1}{2} n_o W}{r_{max}} \tag{6.18}$$

Note the inverse relationship between the number of splats and the size of each splat. Therefore, for materials with large mean free path length and hence large splat size, the number of splats will be small. On the other hand, if the mean free path length is small, the splat size will also be small but the number of splats will be large. Since our algorithm is fill-rate dependent, and the total number of fragments processed is given by the number of splats times the splat size, the rendering time on average for all types of materials with small or large mean free path lengths remains the same.

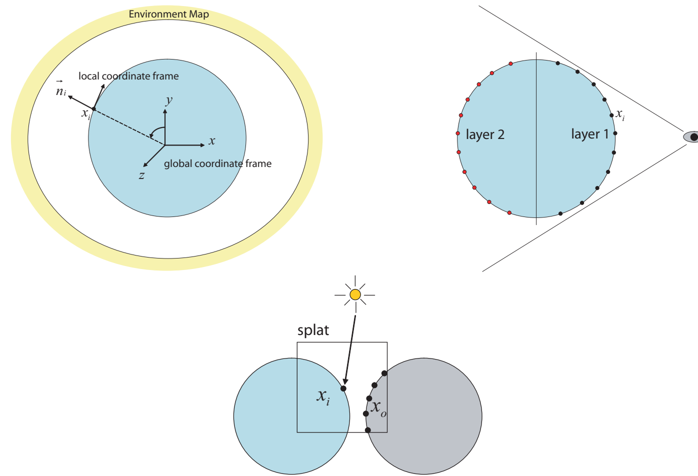


Figure 6.4: (Top) Spherical harmonic coefficients are stored in an environment map after prerotating them into the local coordinate frame formed by the direction vector of the corresponding pixel in the environment map. (Middle) Depth peeling to capture all potential points scattering points, x_i. (Bottom) Since our algorithm operates in image-space, a splat centered at a point x_i on one object can scatter light to receiving points x_o on the other object.

6.4.2 Environment Lighting

So far we have discussed our subsurface scattering algorithm in the context of point light illumination. However, it is easily extended to support environment lighting. There are two key differences: (i) the incident light at each point on the surface of the object has a hemi-spherical distribution instead of just a single illumination direction, and (ii) all points on the surface can potentially receive light as opposed to the point light illumination case in which only the points visible from the light's view can receive light. The first issue is managed using Spherical Harmonics (SH) to represent the environment lighting.

To compute the transmitted light at a given point x_i from the environment, the following integral must be evaluated:

Figure 6.5: From left to right, objects with increasing geometric complexity rendered with subsurface scattering using our algorithm at rates of 26fps, 26fps, and 24fps respectively. Notice the geometric complexity does not greatly influence the render time due to our image-space approach.

$$E(x_i) = \int_{2\pi} L_i(\overrightarrow{\omega_i}) F_t(\eta, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) d\overrightarrow{\omega_i} \quad (6.19)$$

where $L_i(\overrightarrow{\omega_i})$ is the value of the environment lighting function in direction $\overrightarrow{\omega_i}$, and F_t is the Fresnel transmittance term. Combining the cosine term and the Fresnel transmittance function we obtain:

$$E(x_i) = \int_{2\pi} L_i(\overrightarrow{\omega_i}) F_t' d\overrightarrow{\omega_i} \quad (6.20)$$

$$F_t' = F_t(\eta, \overrightarrow{\omega_i})(\overrightarrow{n_i} \cdot \overrightarrow{\omega_i}) \quad (6.21)$$

If both L_i and F_t' are projected into SH basis, then the above integral reduces to a simple dot product operation of their coefficient vectors. The functions L_i and F_t' are projected into SH coefficients which are then pre-rotated for a number of different coordinate frame orientations and stored in environment maps. Therefore for each direction vector $\overrightarrow{\omega}$ corresponding to a pixel in the environment map, the SH coefficients are rotated such that $\overrightarrow{\omega}$ forms the y-axis of the local coordinate frame (see Figure 6.4(Top)). This enables us to lookup the rotated SH coefficients from the environment maps using the normal vector at a particular point. After performing the dot product we obtain the incident light that can then be used in Render Pass

3 of the algorithm. It is important to note that in doing so the angular distribution of the transmitted light is lost. However, multiple scattering can still be computed since the dipole diffuse reflectance function, R_d, depends only on the distance between two points.

The second issue with environment lighting is that of determining the points x_i that receive light. This is handled using depth peeling [Eve01], alternating between the front and back faces at each layer. Figure 6.4(Middle) shows two such layers. The object is rendered multiple times, and in each render pass the depth values from the Z-buffer of the previous pass are used to discard the points that were rendered in that pass. The process is stopped when no points are rendered. Note that this results in a set of textures containing sample points x_i instead of just a single texture in Render Pass 1 of the algorithm (see Section 6.4.1). Therefore, the number of splats required is $n \times n \times l$, where l is the number of layers.

6.4.3 Rendering Multiple Objects

We are able to render multiple translucent objects with different scattering properties simultaneously at no additional cost. The necessary algorithm modifications required for this are explained next.

The first modification involves the dipole texture. Recall that the 1-dimensional dipole diffuse reflectance function, R_d, is stored as a lookup table indexed by the distance. Notice that for each object that we want to render, a separate R_d lookup table must be created. We define a new function that accounts for multiple objects: R_d^{ξ}, where ξ is the object ID. For given set of objects, ξ is simply the index that identifies a given material and retrieves its absorption and scattering coefficients used in the computation of R_d. R_d^{ξ} is therefore

computed and stored in a 2D texture where each row corresponds to the function R_d computed for the object ξ.

The second modification involves storing the object ID as a vertex attribute of the object in order to access the correct row of the dipole texture during the splatting step. This is required because the scattering is performed at a random set of points, x_i, rather than on a per object basis.

If two objects are placed close enough, some splats centered at points on one object may cover the pixels occupied by the second object as shown in Figure 6.4 (Bottom). In such a situation the two objects will contribute light due to multiple scattering to each other. This issue is remedied by storing an object ID along with the 3D world positions for the points x_o in render Pass 2. Then a simple ID check is performed during the multiple scattering computation. If the object IDs of x_i and x_o are the same, multiple scattering computation proceeds as normal. If they are different, the computation is discarded by clipping the pixel.

6.4.4 Comparison with Previous Work

[MKB03a, MKB05]: The key difference between this technique and our algorithm is that we adopt a "splatting" approach to evaluating the area integral in which the scattered light is shot from the light receiving points to the points visible from the camera. By considering only these significant point pairs, the integration computation is performed much more efficiently as opposed to summing up scattering contribution from all the neighboring points. Since our splatting approach does not involve any summation loops, it allows for a more natural implementation on the GPU taking advantage of hardware blending for the accu-

mulation. Furthermore, scattering in back lit objects in the method proposed by Mertens et al. is hindered by the representation of the irradiance. They store the irradiance in a texture from the camera's view, in which case the back lit scattering effect cannot be achieved. Alternatively the irradiance texture can be stored from the light's view to obtain back lit scattering, however this results in artifacts due to stretching when the camera and light views are perpendicular. In a recent update to their algorithm, Mertens et al. propose storing the irradiance in texture space [MKB05]. This allows for back lit scattering since the irradiance field is defined everywhere on the surface of the object. However, as the authors point out, a bijective and continuous parametrization of the surface of the object must be available in order to store the irradiance in texture space. Furthermore, this technique is restricted to non-deformable objects. In our algorithm we use a dual light-camera view approach that enables us to capture back lit scattering in deformable objects and also eliminate stretching artifacts.

[DS03]: Similar to [MKB03a], Dachsbacher and Stamminger [DS03] present a subsurface scattering algorithm that employs translucent shadow maps. The integration is treated as a filtering operation at the visible points where the irradiance from nearby points is weighted and summed to compute the final exiting radiance. In order to speed up the rendering, they approximate the computation by separating the scattering into a local and a global response. At the global level, a heuristically determined filtering pattern is employed that uses the depth and spatial variation between two points to compute the dipole reflectance whereas at the local level the depth variation is ignored. In our algorithm, the distance between the scattering point, x_i, and the visible point, x_o is computed explicitly and therefore the dipole reflectance function is queried accurately. Since Dachsbacher and Stamminger do not provide any comparisons to reference rendering results, it is difficult to verify the accuracy of the images presented in their work. Furthermore, due to GPU

storage limitations they are only able to support directional lighting, whereas our approach manages point light illumination as well as environment lighting.

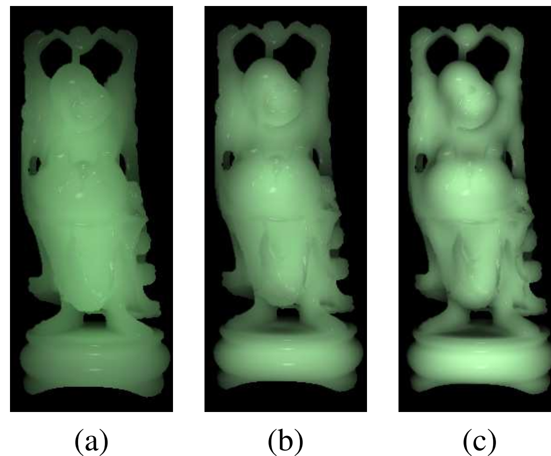
6.5 Results

We implemented our algorithm using HLSL and the Microsoft DirectX 9 SDK. The results presented in this section are from our implementation running on a 2.99 GHz Intel Pentium 4 PC with 1 GB of physical memory and a GeForce 7800 GPU. However, since our algorithm performs no computation on the CPU and does not use any of the system's main memory, a much lower end PC would be sufficient. A shader model 3.0 compatible graphics card is required because our algorithm performs texture lookups in the vertex shader.

Since our algorithm operates in image space, the frame-rate obtained using our method is virtually the same for any object that we render regardless of it's geometric complexity. The only dependence of the speed is on the size of individual splats, the total number of splats, and the resolution of the scatter texture. The number of splats and size of each splat are computed based on the material properties of the object being rendered. In our experiments using error threshold of 0.01 in Equations 16 and 17, the total number of splats ranged from 32×32 for large mean free path lengths (Figure 6.7) to 300×300 for small mean free path lengths (Figure 6.8). To analyze the impact of mean free path length on the performance of our algorithm, we rendered the Happy Buddha model with three different levels of translucency. The results are presented in Table 6.2. As the extinction coefficient increases, the number of splats required increases but the size of each individual splat decreases. Note that although the frame-rate decreases slightly with the increased

number of splats, it remains more or less constant because the decrease in the size of the splats compensates for the extra splats. The relationship of the mean free path length with the total frame-rate and the time taken to render individual splats is show in the graphs in Figure 6.6.

Table 6.2: Rendering performance statistics for translucent materials with varying mean free path lengths.



(a) (b) (c)

Image	FPS	σ_t	number of samples	sample size
(a)	33	0.27	51×51	0.63
(b)	27	1.47	71×71	0.24
(c)	24	3.27	114×214	0.16

On the average, with a scatter texture resolution of 800×600 pixels we are able to obtain rates of around 25 frames per second. Figure 6.5 shows a sequence of images of objects with increasing level of geometric complexity rendered using our algorithm. Notice that the frame-rates are consistent, regardless of the object geometry. The small differences in the frame-rates are due to the rasterization overhead in the intermediate render passes of the algorithm.

Figure 6.7 demonstrates our algorithm's ability to render subsurface scattering in back lit objects. The increase in attenuation of the scattered light in the stomach and thigh regions of the horse gives the perception of depth, which is characteristic of back lit translucent objects. Since our algorithm has no pre-computation

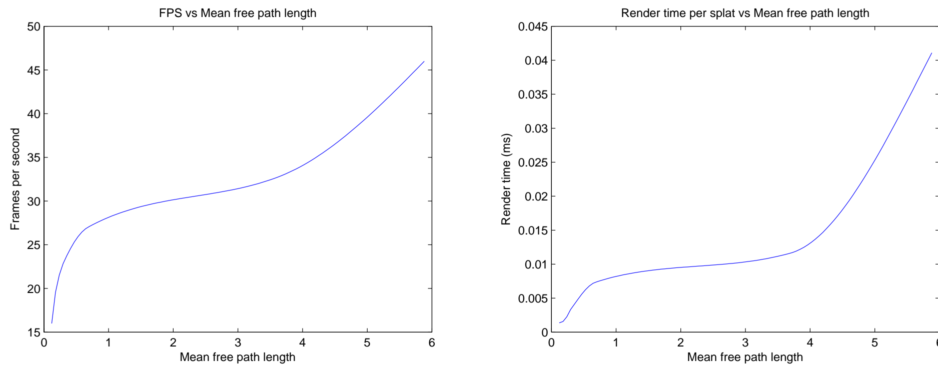


Figure 6.6: (Top) Graph showing the relationship between the mean free path length and the rendering frame-rate of the algorithm. As the mean free path length gets larger, the number of splats decreases and therefore the frame-rate increases. (Bottom) Graph showing the relationship between the mean free path length and the time taken to render individual splats. As the mean free path length gets larger, the splat size increases, and therefore takes more time to render.

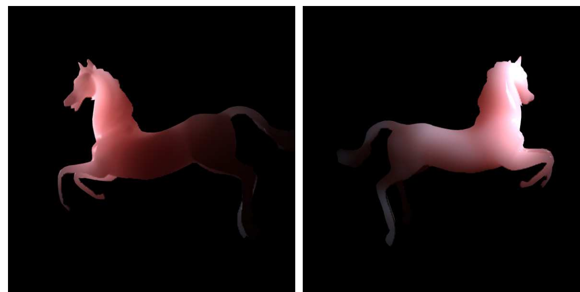


Figure 6.7: Subsurface scattering in a back lit (left) and front lit (right) horse model. These images were rendered at 24fps at a resolution of 800×600 pixels.

and is evaluated fully at every frame, it gives us the freedom to dynamically change lighting, viewing, and geometry. The supplementary video shows an animation of subsurface scattering in a deformable object. In this experiment, the 3D teapot model is dynamically displaced to show the effect of change in geometry on the scattering behavior. Another interesting feature of our approach is that it allows us to render multiple objects with different scattering properties simultaneously at no additional cost. Figure 6.9 shows images of two squirrel models with different absorption and scattering coefficients rendered at 25 fps. It also demon-

strates the clipping technique explained in Section 6.4.3 to prevent scattering from one object to another. Figure 6.12(Bottom) shows results from our algorithm with environment lighting.

Our algorithm effectively manages shadowing from other occluding objects as well as self shadowing. Figure 6.8 shows images of a human head model rendered using our algorithm. Note the blurring of the shadows cast by the teapot and the ear due to subsurface scattering. Self shadowing is obtained implicitly without any additional steps, whereas shadows from other objects require a shadow map lookup to determine irradiance at splat locations, x_i. The images in Figure 6.8 were rendered at 23 fps. A video clip of this rendering is included in the supplementary video.

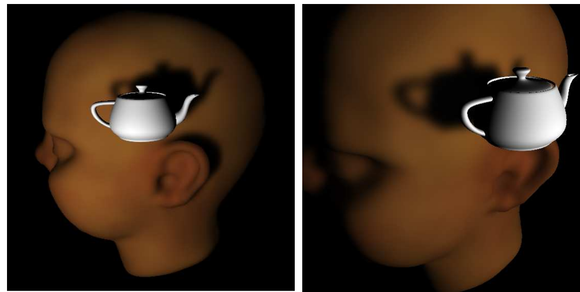


Figure 6.8: (left) Shadows cast from external and self occlusion. (right) Close-up of the blurred shadow due to scattering.

Donner and Jensen [DJ05] recently introduced a multipole diffusion approximation for computing scattering in layered materials. This new multipole function can be substituted for the dipole function in order to render objects made up of layers with different scattering properties. Note that supporting the multipole approximation does not affect our algorithm's real-time performance; it only influences the cost of creating the lookup table. We demonstrate results using the multipole method in our rendering system in Figure 6.12(Top).

6.5.1 Analysis

As shown in Figure 6.1, our algorithm produces results comparable to those obtained using offline renderers. The main difference between our method and most other existing techniques that employ the dipole BSSRDF model is that we compute the area integral for multiple scattering in image-space. This is done by rendering quads at points x_i on the surface of the translucent object that is exposed to the light source. The quads provide access to points x_o visible to the camera, to which the light is scattered from each x_i. As is the case with all other methods, the sampling rate for x_i and x_o must be sufficient to prevent any visual artifacts. In our algorithm, the number of x_o points is determined by the resolution of the image being rendered, whereas the number of x_i points is determined by the scattering properties of the material and thresholds chosen by the user as explained in Section 6.4.1. Since the number of points x_i corresponds to the number of splats rendered, it directly influences the rendering speed. Choosing a large threshold will result in too few x_i and produce patchy artifacts as shown in Figure 6.10, whereas an excessively conservative threshold will result in a large number of x_i, hence increasing the render time. We used a threshold of 0.01 (1% Weber threshold) in our implementations.

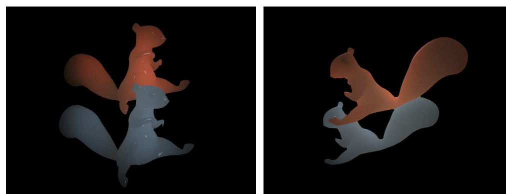


Figure 6.9: Multiple objects with different scattering properties rendered at 25fps. Note that due to our image-space clipping technique, scattering from one object to the other is prevented even though the objects are adjacent.

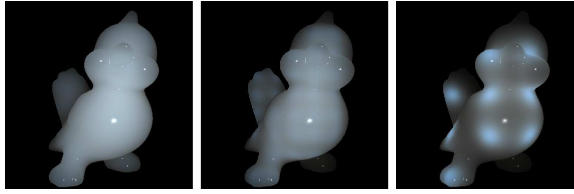


Figure 6.10: From left to right: the bird model rendered using 32×32 samples at 31 fps, 16×16 samples at 65 fps, and 8×8 samples at 151 fps. Undersampling results in "patchy" artifacts because the splats do not overlap sufficiently.

As a direct consequence of the sampling strategy used to select the points x_i, temporal artifacts are introduced in certain cases. Recall that x_i correspond to the pixels of the render target texture on which the 3D world positions of the object have been rendered from the light's view (see Section 6.4.1, render Pass 3). Pixels that fall on the surface of the object get converted into sample points x_i, and the blank pixels are discarded. Therefore, the number of points x_i is in fact dependent on the orientation and positioning of the object as observed from the light's view. Thus, for example, when the object rotates, some flickering can be noticed due to the changing number of x_i. The amount of flickering also depends on the shape of the object being rendered. An object with fairly skewed dimensions is more likely to cause temporal artifacts as opposed to a symmetric object, such as a sphere, in which the artifacts are negligible. To reduce the appearance of these artifacts, we employ a sampling strategy that maintains sampling coherence in between frames. This technique is explained in the following subsection.

6.5.2 Reducing Temporal Artifacts

In order to reduce flickering caused by the change in the apparent projected area of the translucent object with respect to the light's view, sampling coherence in between frames must be maintained. Recall that

for each frame, the object is rendered from the light's view to obtain splat locations. Thus every frame, a completely new and independent set of splat locations is obtained. If the sampling rate is not high enough, this results in visible discrepancy in the final rendered images. In order to enforce coherence between frames, we modify our splat location sampling strategy to reuse certain samples from previous frames. At each frame, the light view positions textures from both the current and the previous frame are kept. The $n \times n$ splats are rendered twice, first using the positions texture from the current frame and then using the positions texture from the previous frame. However, all samples from each of the two sampling grids are not splatted. Splat locations are dynamically selected, discarding the unwanted samples by clipping them in the vertex shader. The steps of this sampling strategy are as follows:

Pass 1: Render $n \times n$ splats using light view positions from current frame

For each sample, x_i:

i) Project x_i into the light's view of the previous frame and look up the previous positions texture to obtain x_i'.

ii) If distance(x_i, x_i') $< \varepsilon$, then clip x_i. Else, splat x_i

Pass 2: Render $n \times n$ splats using light view positions from previous frame

For each sample, x_i':

i) Project x_i' into the light's view of the current frame and look up the current positions texture to obtain x_i.

ii) If depth of x_i' is greater than depth of x_i, then clip x_i'. Else, splat x_i'.

With this sampling technique, all the samples from the previous frame that sampled a particular region of the surface of the object visible from the light source are retained in the current frame, hence enforcing sampling coherency in between frames. Note that when the object doesn't change its orientation with respect to the

light's view, the previous and current frame's position texture is the same. In that case if the above sampling technique is used, all the samples in Pass 1 will be discarded and all the samples in Pass 2 will be splatted. When the object moves with respect to the light, all the samples in Pass 2 are splatted except the ones that are no longer visible from the light's view in the current frame.

This new sampling strategy incurs some additional vertex processing cost since two $n \times n$ splatting grids are processed instead of just one. However, the bottleneck of the algorithm is the fill-rate and not the vertex processing. The total number of samples that actually get splatted remain close to $n \times n$ due to the clipping of unwanted samples. Therefore, the frame-rate is not significantly affected.

6.6 Conclusion and Future Work

We presented a real-time image-space algorithm for rendering translucent objects using dipole as well as multipole BSSRDF model. The algorithm computes the area integral using a splatting approach in contrast to the traditional gathering approach used by existing methods. The splatting approach allows us to implement the algorithm entirely on the GPU. The algorithm employs a dual view representation (light view and camera view) for storing the incident light and performing the scattering operation, which enables us to account for all the subsurface scattering effects while avoiding artifacts associated with space conversion. Furthermore this approach is also efficient since it only performs the scattering computation between significant point pairs (emitting and receiving), thus eliminating any unnecessary calculations. Our method is able to produce visually convincing results at rates of about 25 fps for arbitrarily complex objects under dynamic view and illumination.

In future we would like to accommodate translucent objects with heterogeneous subsurface scattering properties in our current real-time rendering system.

Figure 6.11: The Stanford Bunny model made of a highly absorbing material. Notice how the back lighting effect at the silhouettes rapidly decreases toward the center of the model due to the high material density. This image was rendered at 22fps.

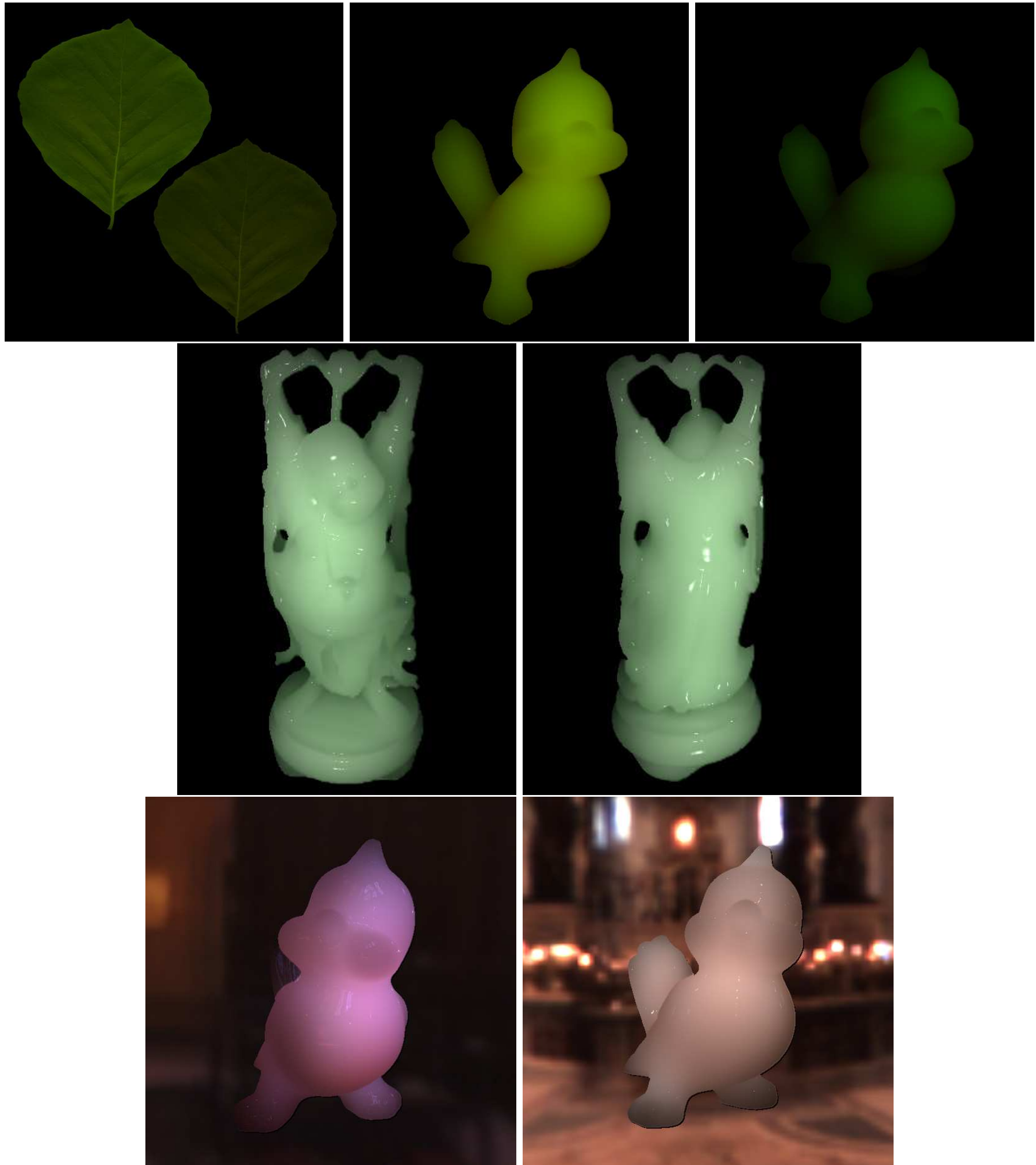


Figure 6.12: (Top) Multilayered material rendered using the multipole diffusion method with our algorithm. The material is made of a thin yellow layer and a green layer. The second and third images show the effect of increasing and decreasing the thickness of the yellow layer. (Middle) The Happy Buddha model rendered using our algorithm with subsurface scattering properties of jade. (Bottom) The bird model rendered using our algorithm under environment lighting: (left) Grace Cathedral and (right) St. Peters.

CHAPTER 7

ARTIST DRIVEN INTERACTIVE EDITING OF HETEROGENEOUS TRANSLUCENT MATERIALS

7.1 Abstract

In this work, we present a simple model for representing heterogeneous translucent materials that is suitable for artistic editing. The model is based on the dipole diffusion approximation for multiple scattering in homogeneous translucent materials, augmented with spatially varying modifier functions to simulate heterogeneous scattering. These functions are defined by the artist via brush strokes, enabling them to paint translucency directly on the 3D object. Our editing system utilizes previous work on interactive rendering of translucent materials to provide real-time updates, but can also be used in off-line renderers for high quality image synthesis.

7.2 Introduction

Recent advancements in acquisition and representation of the surface reflectance properties of translucent materials have provided opportunities of accurate visualization of a larger class of materials in computer graphics. Translucent materials, unlike opaque materials, exhibit subsurface scattering of the incident light. Light impinging the surface of a translucent object penetrates the surface and undergoes multiple scattering events. Therefore, translucent materials cannot be represented accurately using BRDFs (Bidirectional Reflectance Distribution Functions), which assume that the incident light is either reflected or absorbed at the point of incidence. The extra possibility of the light being scattered, in addition to the reflection and absorption events, is also required for translucent materials. The scattering of light adds an additional spatial distribution to the reflectance field, and is represented by the BSSRDF (Bidirectional Surface Scattering Reflectance Distribution Function), along with the angular distribution. The BSSRDF is thus an 8 dimensional function: $S(x_i, x_o, \omega_i, \omega_o)$, which gives the light scattered from a point x_i from direction ω_i to a point x_o in direction ω_o. In order to compute the final outgoing radiance at a point x on the surface of a translucent object, the light scattered towards x from all points x_i on the surface must be accumulated.

Translucent materials can be represented and rendered using discrete BSSRDFs, obtained through measurements of real world material samples. However, it is desirable to analytically model these materials in terms of physical properties. Jensen et al. developed such a model, called the dipole diffusion model [JML01], for highly scattering, homogeneous translucent materials that computes the scattering response based on physical parameters such as the absorption and scattering coefficients. This model was derived for materials in which the scattering of light dominates all other events as the light passes through the medium. Due to this, the radiance field loses its directionality and becomes a diffuse distribution. Translucent materials,

such as marble, exhibit this property. However, other types of translucent materials that do not exhibit strong multiple scattering appear more transparent, such as colored frosted glass. This is because the incident light maintains its direction of propagation as it traverses the material.

Although the dipole diffusion model provides a simple and elegant representation for reflectance of translucent materials, it only applies to materials with homogeneous scattering properties. Most of the interesting materials, both translucent and non-translucent, are rarely perfectly homogeneous across the entire object. With the exception of certain materials such as milk, most materials have spatial variations, like the black veins on a white marble tile. Such materials cannot be modeled by a simple dipole diffusion model. In this paper, we show how the dipole model can be augmented with spatially varying modifier functions to represent heterogeneous translucent materials. We employ this model in an artistic tool for creation and editing of heterogeneous translucent materials. In essence, we provide a simple interface to the artist so that he or she can directly *paint translucency* on a 3D object using brushes that the artist is already familiar with.

7.2.1 Editing translucent materials

Creation and editing of translucent materials plays an important part in many computer graphics related industries. In games and computer-generated movies, great artistic care is given to the material design for the primary cast of characters. This cast usually features mainly humans and animals, organic objects whose material appearance is largely dominated by scattering of light underneath the surface, rather than simple surface reflectance. In order to produce this translucency effect, the content creators generally employ certain artistic techniques in terms of visual cues to obtain the desired appearance. However, such artistic

freedom can cause problems during rendering, which generally involves physically based lighting computation. Thus in order to comply with existing light transport algorithms, the materials should be defined in terms of physically meaningful parameters of a scattering model. However, these parameters are usually not meaningful to an artist, who thinks in terms of visual effects, like blur and glow. Unless one intimately understands the behavior of a given scattering model, such as the dipole diffusion model, it is not immediately clear what parameters to change to achieve a specific visual effect. Material design thus can often become a tedious cycle of guess-render-repeat.

In this paper, we analyze the dipole diffusion model and develop an editing tool for translucent materials that can be used by artists. We observe the visual effects resulting due to changes in the various physical parameters of the dipole model, and provide a simple and intuitive way of recreating certain effects without having to deal with adjusting the model parameters explicitly. In order to maintain the underlying dipole model, the changes in the physical parameters of the model corresponding to the artist edits must be computed. However, this requires expensive non-linear optimization, which is not feasible for interactive applications. Instead, we employ two spatially varying modifier functions that augment the dipole diffusion function to produce the same results without modifying the dipole parameters. These modifier functions are computed directly as the artist paints the appearance of the heterogeneous translucent object. We employ our real-time rendering algorithm for translucent materials from our previous work (see previous chapter) to provide interactive visual feedback to the user. Since our heterogeneous scattering model is based on the dipole diffusion model, it can also be used in off-line renderers for high quality image synthesis, which is required in computer-generated movies.

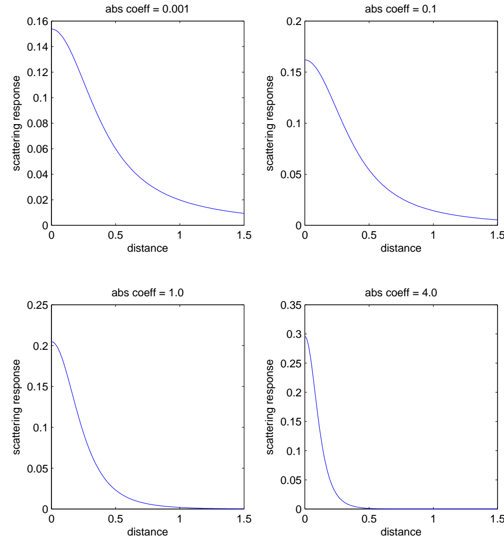


Figure 7.1: The effect of increasing the absorption coefficient, σ_a, on the scattering response of the dipole diffusion function. The other parameters used to create these graphs are: $\sigma_s = 2.0$, $g = 0.0$, $\eta = 1.2$. As the absorption coefficient increases, so does the total extinction coefficient, and the function falloff becomes sharper indicating that the effective scattering range is decreasing rapidly.

7.3 Analysis of the dipole diffusion model

The dipole diffusion model provides an approximation to the scattering reflectance distribution in highly scattering homogeneous translucent materials. This distribution is radially symmetric, and thus depends only on the distance between the scattering point, x_i, and the receiving point, x_o:

$$R_d(r) = \frac{\alpha'}{4\pi}[z_r(\sigma_{tr} + \frac{1}{d_r})\frac{e^{-\sigma_{tr}d_r}}{d_r^2} + z_v(\sigma_{tr} + \frac{1}{d_v})\frac{e^{-\sigma_{tr}d_v}}{d_v^2}] \tag{7.1}$$

The terms used in Equation 7.1 are described in Table 6.1. The BSSRDF given by the dipole model is defined in terms of the diffusion function, R_d, as follows:

$$S(x_i, x_o, \omega_i, \omega_o) = \frac{1}{\pi} F_t(\eta, \omega_i) R_d(||x_i - x_o||) F_t(\eta, \omega_o) \tag{7.2}$$

where F_t is the Fresnel transmittance function, and η is the refractive index of the material. In order compute the outgoing radiance at a point x_o for rendering, the scattering contribution to x_o from all points x_i on the surface of the object must be accumulated:

$$L_o(x_o, \omega_o) = \frac{F_t(\eta, \omega_o)}{\pi} \int_A R_d(||x_i - x_o||) \int_{2\pi} L_i(x_i, \omega_i) F_t(\eta, \omega_i)(n_i \cdot \omega_i) d\omega_i dA(x_i) \tag{7.3}$$

Our goal is to analyze how changes in the physical parameters of the dipole model affect the final outgoing radiance, and hence the visual appearance of the rendered result.

The dipole diffusion function, R_d, is defined in terms of four physical parameters of translucent materials: absorption coefficient (σ_a), scattering coefficient (σ_s), anisotropy term of the Henyey-Greenstein phase function (g), and the refractive index (η). Once these parameters are fixed, R_d is simply a 1-dimensional function of the distance between two points. We begin our analysis by changing one physical parameter of the dipole model at a time, while keeping the other three fixed, and plotting a graph of the function to study the effect of the change on the scattering response. The results are presented in the following subsections.

7.3.1 Changing σ_a and σ_s

The absorption and scattering coefficients, σ_a and σ_s, are collectively referred to as the extinction coefficient: $\sigma_t = \sigma_a + \sigma_s$. The extinction coefficient determines the total attenuation of light due to absorption and scattering. As σ_t increases, the effective scattering range decreases because the light signal gets attenuated faster. Therefore, for materials with high values of σ_t, the scattering range is very small and the entire scattered light energy is concentrated in a very small local neighborhood. Figure 7.1 shows graphs of the dipole diffusion function response plotted against the distance for different values of σ_a. Note that as σ_a is increased, the falloff of the curve becomes steeper.

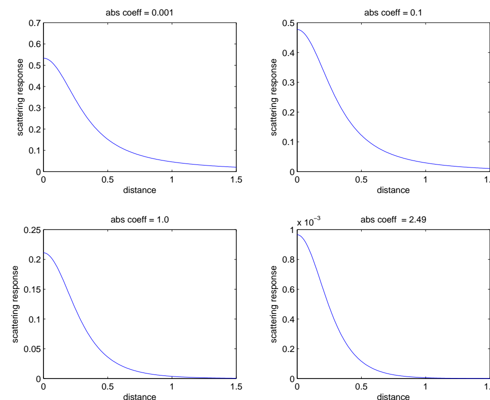


Figure 7.2: The effect of increasing the absorption coefficient, σ_a, on the scattering response of the dipole diffusion function, while adjusting the scattering coefficient to keep the extinction coefficient fixed at 2.5. Other parameters have also been fixed at same values: $g = 0.0$, $\eta = 1.2$. Note that increasing the absorption coefficient decreases the overall scattering response.

Increasing the scattering coefficient, σ_s, also increases the extinction coefficient, and therefore has a similar effect on the scattering range. To isolate the effects of σ_a and σ_s on the scattering response, we fix the extinction coefficient between different plots. Figure 7.2 shows the graphs with increasing absorption coefficient, where the scattering coefficient has been adjusted to fix the extinction coefficient at 2.5 for all

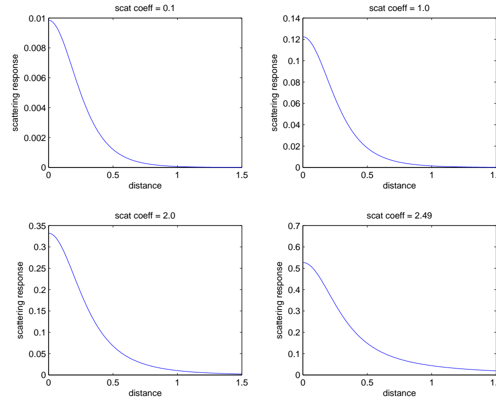


Figure 7.3: The effect of increasing the scattering coefficient, σ_s, on the scattering response of the dipole diffusion function, while adjusting the absorption coefficient to keep the extinction coefficient fixed at 2.5. Other parameters have also been fixed at same values: $g = 0.0$, $\eta = 1.2$. Note that increasing the scattering coefficient increases the overall scattering response.

four plots. It can be observed that the maximum response of the dipole diffusion function decreases as the absorption coefficient increases. Note that the falloff of the curve does not change much since the total extinction coefficient is fixed. Similarly, Figure 7.3 shows the plots of scattering response with increasing scattering coefficients. Again, the extinction coefficient has been fixed to 2.5 by adjusting the absorption coefficient accordingly. Increasing σ_s also increases the maximum response of the scattering function.

It can be concluded that changing the absorption and scattering coefficients effect the sharpness of the falloff as well as the magnitude of the scattering response. The total extinction coefficient is responsible for the sharpness of the falloff, and hence the effective scattering range, the absorption and scattering coefficients individually are responsible for decreasing and increasing the magnitude of the scattering response.

7.3.2 Changing g

The anisotropy term, g, determines the directional scattering bias of the translucent material under consideration. The values of this term range from -1 to 1, where $g = -1$ indicates total backward scattering, $g = 1$ indicates total forward scattering, and $g = 0$ indicates isotropic scattering. The dipole diffusion model is derived for semi-infinite, planar objects, thereby placing both the light source and the viewer on the same side of the surface. This implies that if the material is totally forward scattering, that is if $g = 0$, then no light will be scattered back to the surface and hence the scattering response will be 0. Therefore, only those materials that exhibit some backward scattering contribute to the scattering response in the dipole diffusion model. This theory was verified by plotting the scattering response graph with varying values of g (see Figure 7.4).

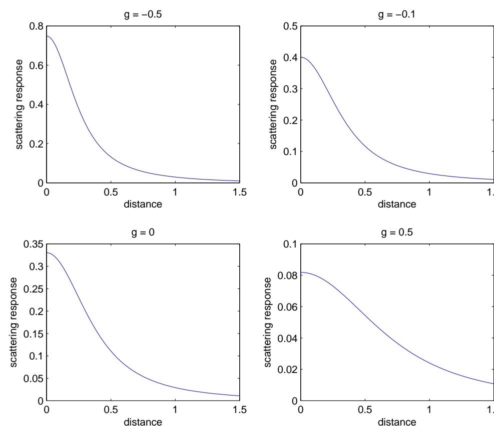


Figure 7.4: The effect of changing the anisotropy term, g, on the scattering response of the dipole diffusion function. Other parameters have been fixed at the following values: $\sigma_s = 2.0$, $\sigma_a = 0.1$, $\eta = 1.2$. Note that as g increases, indicating increased forward scattering, the total scattering response decreases.

The anisotropy term also effects the steepness of the function because it modifies the scattering coefficient: $\sigma_s' = (1-g)\sigma_s$, where σ_s' is called the effective scattering coefficient. For negative values of g, the effective

scattering coefficient increases and so does the extinction coefficient, thereby making the falloff of the scattering response curve sharper. Note that when $g = 0$, the effective scattering coefficient is the same as the scattering coefficient.

7.3.3 Changing η

The refractive index accounts for the change in the transmitted irradiance at the boundary interface between the translucent material and air. Increasing the refractive index, which indicates light moving from a coarser medium to a more dense one, decreases the transmitted irradiance and therefore also the scattering response (see Figure 7.5).

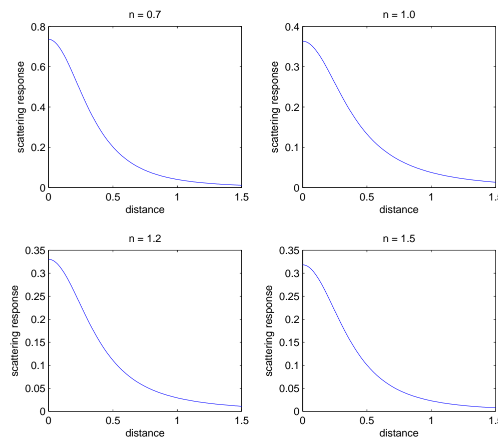


Figure 7.5: The effect of changing the refractive index, η, on the scattering response of the dipole diffusion function. Other parameters have been fixed at the following values: $\sigma_s = 2.0$, $\sigma_a = 0.1$, $g = 0$. Note that as η increases, indicating light propagation from coarser to denser medium, the total scattering response decreases.

In the following section, we will look at the visual implications of the various properties of the scattering response graphs. We will then formulate a mapping between the physical scattering parameters of the dipole diffusion model and the visual effects.

7.3.4 Perceived visual effects of scattering

From the analysis in the previous section, it can be observed that changing the parameters of the dipole model result in two basic different effects on the scattering response: (1) change in the falloff of the function, and (2) change in the overall magnitude of the function. All types of changes in the dipole model affect one or both of these properties. Therefore, in order to understand the perceived visual effects of changes in σ_a, σ_s, g, and η, we must study how the falloff and magnitude of the scattering function influence the irradiance signal to render the final image.

A scattering function with a very smooth falloff implies that the effective scattering range is large. This means that the irradiance at a given point is scattered to other points on the surface at larger distances, hence creating a blurred final image. On the other hand, if the falloff of the scattering function is sharp, the resulting image will be less blurry. Figure 7.6 shows the effect of change in the falloff of the scattering function on the rendered image. The images in the figure have been rendered by applying the dipole diffusion function on a checkerboard irradiance function to observe the visual effect. Based on our findings in the previous section, we make the scattering function's falloff smoother by decreasing the extinction coefficient. As the extinction coefficient gets smaller, the spread of the scattering function gets larger, resulting in blurrier images due to long distance scattering.

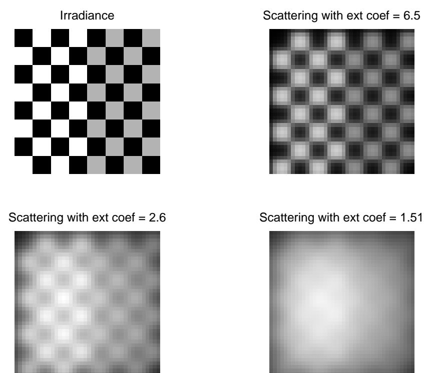


Figure 7.6: The effect of decreasing the extinction coefficient (thus increasing the smoothness of the falloff of the scattering function) on the irradiance. As the extinction coefficient decreases, the effective scattering range increases. This results in the irradiance field getting more and more blurry.

The change in the magnitude of the scattering function simply increases or decreases the brightness of final image. Figure 7.7 shows the results observed by increasing the scattering coefficient, while adjusting the absorption coefficient to maintain the same extinction coefficient as the corresponding images in Figure 7.6. Note that although Figure 7.7 is brighter than 7.6, both Figures 7.6 and 7.7 have more or less the same blurry appearance.

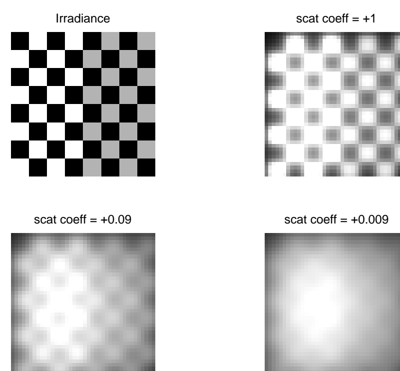


Figure 7.7: Increasing the scattering coefficient, while fixing the extinction coefficient, makes the final rendered image brighter. Note the numbers each image is the increase in the scattering coefficient with respect to the corresponding image in Figure 7.6

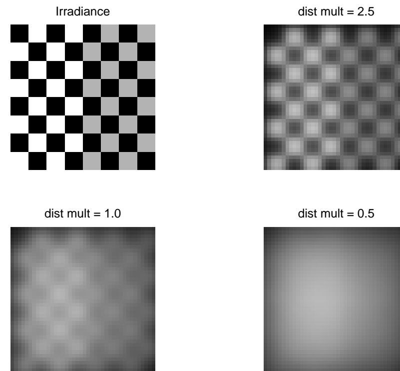


Figure 7.8: Increasing the blur in the final rendered image by scaling up and down the distance input to the dipole diffusion function has the same effect as modifying the extinction coefficient.

7.4 Artist driven model for heterogeneous translucent materials

Thus far we have shown that the physical parameters of the dipole diffusion model modify the sharpness of the falloff and the magnitude of the scattering response. We have also shown that these two properties translate to two unique visual effects: amount of blur, and brightness or glow of the final image. An artist can easily identify with these effects, and therefore they serve as good tools for artist driven creation of translucent materials. Our editing system allows an artist to modify a translucent material using blur and brightness *brushes* that can be applied directly on a 3D object. Using these brushes, the artist can make certain parts of the object more or less translucent than others by applying different amounts of blur at different regions. Similarly, the artist can also increase or decrease brightness at certain regions. This creates a non-homogeneous distribution of scattering properties across the surface of the material

Providing such an editing system in terms of blur and brightness modification tools that still conforms to the dipole diffusion model would require fitting for the four physical parameters varying spatially over the entire surface. However, this task is very difficult because it involves non-linear optimization. Furthermore,

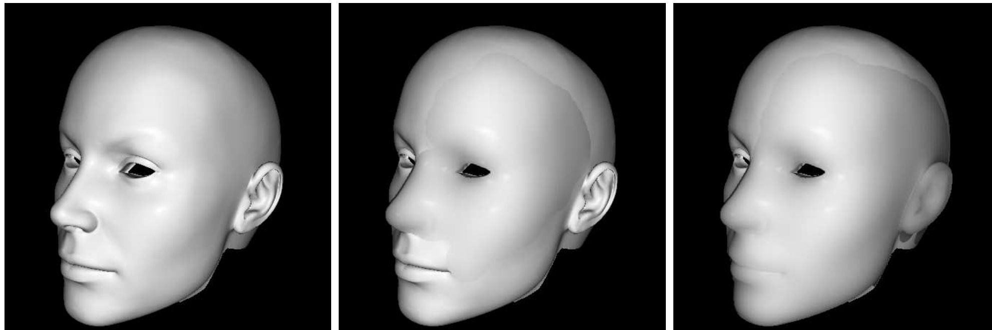


Figure 7.9: The artist starts with a non-translucent surface and starts painting translucency directly on the 3D mesh. Note the soft and blurred appearance of the painted regions.

due to the under-constrained nature of the problem, a unique solution is not likely. Therefore, we employ an alternative approach to mapping the effect of the intuitive blur and brightness brush strokes into dipole diffusion function parameters, involving the use of a spatially varying distance multiplier function.

7.4.1 Augmented dipole diffusion model for heterogeneous scattering

The basic idea behind our augmented dipole model is to modify the distance argument to the diffusion function rather than the physical scattering parameters of the material. When an artist applies a blur brush at a particular region, the scattering function at that region increases its spread to cover a larger distance. This can be accomplished by scaling down the distance input to the diffusion function with a fixed set of scattering parameters. Therefore, scaling the distance down and up has the same result as making the region more and less blurry respectively. Figure 7.8 shows images rendered using fixed dipole parameters with varying distance multiplier values.

The amount of blur is inversely proportional to the distance multiplier required to produce that blur. Modifying the distance input in such a manner is both effective and intuitive

The brightness brush, which increases the glow at the applied regions, is represented using a spatially varying function that is multiplied with irradiance. This function therefore simply scales the irradiance down or up, resulting in the image being less or more bright respectively.

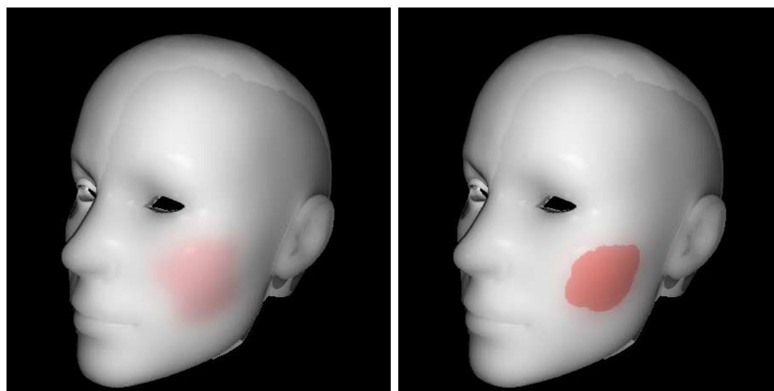


Figure 7.10: Two different effects created with varying amount of blur.

The spatially varying distance multiplier function and irradiance modulation function, applied to the dipole diffusion model forms our new augmented model for representing heterogeneous translucent materials. The dipole BSSRDF in Equation 7.2 is modified as follows:

$$S(x_i, x_o, \omega_i, \omega_o) = \frac{M_i(x_i)}{\pi} F_t(\eta, \omega_i) R_d(||x_i - x_o|| * M_d(x_i)) F_t(\eta, \omega_o) \tag{7.4}$$

where M_i is the irradiance modulation function and M_d is the distance multiplier function. Note that these functions are defined at each point, x_i, on the surface of the object. Therefore different regions can have different scattering distributions, which is the case in heterogeneous translucent materials. We present some results from out editing system in the next section.

Figure 7.11: Painting with a skin colored brush. This modifies the irradiance modulation texture, which controls the magnitude of the scattering response. The desired colored scattering effect is thus created by adjusting the scattering magnitude in each of the R, G, and B channels separately.

7.5 Results

We implemented our dipole diffusion based heterogeneous scattering model in an editing tool to create non-homogeneous translucent objects. The tool allows the artist to paint the spatially varying functions discussed in the previous section, which are stored as texture atlases. We employ three different textures in our implementation: distance multiplier texture, irradiance modulation texture, and final outgoing radiance modulation texture. The first two textures account for the different scattering properties of the translucent material, and the third texture is used to differentiate between the translucent and non-translucent regions of the object. The regions influenced by this texture do not undergo subsurface scattering, and therefore appear non-translucent.

Our editing system starts off by loading an arbitrary 3D model and rendering it as a fully opaque, Lambertian diffuse surface. The user then starts painting translucency directly on the 3D mesh by using the translucency brush. Figure 7.9 shows frames from a typical painting session. The regions that the user touches with the brush are affected by subsurface scattering, and hence appear translucent. This creates a heterogeneous

distribution of scattering and non-scattering regions on the surface of the object. The scattering behavior of the scattering regions can be further modified using the blur brush, which controls the effective scattering range of the translucent material. Figure 7.10 shows how the artist can use this brush to create two unique effects. In the first image, the user paints a pink region in the cheek area of the model with a large amount of blur to create the appearance of rosy cheeks. In the second image, the blur has been reduced to minimum to create the appearance of pink paint applied on top of the cheek. The color of these painted regions is stored in the irradiance modulation texture. Recall that this function controls the magnitude of the scattering response. By adjusting the magnitude differently in each of the three color channels, R, G, and B, colored scattering effects such as the ones shown in Figure 7.10 can be obtained. Figure 7.11 shows this same technique applied for creating a skin-like appearance.

Our editing system utilizes our image-space rendering algorithm for translucent materials. The frame-rates of the editing system match closely to those produced by the original rendering algorithm for homogeneous translucent materials. This is because the rendering algorithm allows for full evaluation of the dipole model for every rendered pixel at interactive frame-rates. Hence, the only overhead imposed by the editing system is that of maintaining three extra textures per object. Therefore, since the fill-rate remains the bottleneck, there is no significant drop in the frame-rate.

It is important to note that our heterogeneous scattering model maintains the underlying dipole diffusion model. Due to this, existing physically-based, high quality rendering systems that utilize the dipole model for rendering translucent materials can also use our new model for rendering heterogeneous translucent materials with minor modifications. This is not the case with ad hoc models that are purely artist driven and have no physical basis.

7.6 Summary and future work

We have presented an artist driven editing system for creating and rendering heterogeneous translucent materials. By analyzing the effect of change of the dipole parameters on the scattering response, we showed that all variations in the scattering properties of a material can be represented by adjusting the spread or the smoothness of the falloff of the scattering response function, and the magnitude of the scattering response function. Studying the visual effects of these two properties, it was observed that the falloff of the function controls the amount of blur in the rendered image, whereas the magnitude of the function controls the brightness or glow in the rendered image. Therefore, we developed our editing system based on these intuitive two tools: blurring and changing brightness. However, mapping the effect of these tools into physical parameters of the dipole diffusion model requires non-linear optimization of an under-constrained problem, which is difficult and not feasible for an interactive application. We showed that this can be avoided by keeping the dipole parameters fixed, and augmenting it with modifier functions that are easy to manage and allow us to produce the same results. Two modifier functions were employed, one for modulating the distance input to the dipole function, and the other for modulating the irradiance. The former controls the falloff of the scattering response function, whereas the latter controls the magnitude. By defining these functions over the entire surface of the object, the scattering at each point can be uniquely controlled, thereby creating a heterogeneous scattering distribution.

In this paper, we have provided a tool for an artist to create a heterogeneous translucent object. However, it is also desirable to extract the scattering properties of a heterogenous translucent material from photographs and render 3D objects using those properties. In future work, we would like to address this problem. Specifically, we would like to extract the spatially varying distance multiplier function, the irradiance modulation

function, and base scattering parameters of the dipole diffusion model, such that they can be directly plugged into our current system for further editing. The ability to extract scattering properties from images along with manual editing would provide a complete editing solution for heterogeneous translucent materials.

CHAPTER 8

SUMMARY AND FUTURE WORK

In this thesis, we have shown that certain problems in real-time realistic rendering can be efficiently solved in the image-space rather than the geometric-space. We have targeted three main problems: (i) representing and illuminating large scenes, (ii) rendering of complex indirect illumination effects, and (iii) subsurface scattering. The first problem was observed in the context of grass rendering, which involves processing large amounts of geometry, representing the individual grass blades, and also computing lighting at each of the blades. We showed that patches of grass can be more efficiently represented using a set of pre-illuminated images than geometry by developing a rendering algorithm employing BTFs. Our algorithm produces results at interactive frame-rates and is only fill-rate dependent; that is, the execution speed only depends on the number of pixels in the rendered image and not the number of grass blades in the scene.

For the second problem, we developed an algorithm for rendering caustics. The main issue in real-time caustics rendering is that it involves computing intersections of refracted or reflected light rays with objects in the scene. In our work, we showed that these expensive intersection computations can be approximated by reformulating the problem as that of finding the root of a discrete function representing the height field of the 3D scene. The problem is then easily solved using an iterative numerical method. Our rendering results

show that our algorithm produces visually convincing caustics at interactive frame-rates that are practical enough for use in games and simulation systems.

Next, we presented an algorithm for interactive rendering of subsurface scattering in translucent materials. In this work, we described how a dual light-camera view representation can be used to efficiently solve the area integral for scattering in image-space. By obtaining the points on an object that are visible to the light source, and the points that are visible to the camera, through rasterization, we are able to identify significant point pairs that influence the illumination due to scattering. In doing so, the total number of computations are reduced significantly, thus resulting in faster execution. Our algorithm produces accurate results at interactive frame-rates, and are visually comparable to those produced using off-line rendering.

Lastly, we utilized our rendering algorithm for translucent materials to develop an artist driven editing system for creating heterogeneous translucent objects. In this work, we start by providing an in-depth analysis of the dipole diffusion model and show the visual effects corresponding to the changes in the physical scattering parameters of the model. This enabled us to create intuitive tools that the artist can use to modify translucency of a material without having any knowledge about the underlying diffusion model. We showed that by augmenting the dipole model with spatially varying modifier functions, we are able to create a heterogeneous distribution of subsurface scattering. Furthermore, this also allows us to provide a direct mapping between the artist tools and the effect on the scattering response without changing the parameters of the dipole model.

For future work, we would like to further extend our work on real-time subsurface scattering. In particular, we would like to address the limitations of the dipole source approximation for the diffusion of light due to multiple scattering. The dipole model is designed with the assumption that the subsurface scattering is being

computed for a semi-infinite and planar object. In practice, however, it is applied to objects of all shapes and sizes, even though in that case the model is theoretically invalid. Therefore, we would like to study the effect of shape and size on the diffusion function by measuring the scattering response in real-world objects. By analyzing this data, we would be able to determine the error in the dipole model due to the violation of the size and shape assumptions. Furthermore, this would allow us to more accurately model the diffusion function in terms of shape and size parameters, in addition to the scattering properties of the material.

LIST OF REFERENCES

[Arv86] James Arvo. "Backward Ray Tracing." In *Developments in Ray Tracing, SIGGRAPH '86 Course Notes*, August 1986.

[BAS02] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. "Shadow Mapping for Hemispherical and Omnidirectional Light Sources." In *Computer Graphics International*, 2002.

[Bli78] James F. Blinn. "Simulation of wrinkled surfaces." In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pp. 286–292, New York, NY, USA, 1978. ACM Press.

[BN76] James F. Blinn and Martin E. Newell. "Texture and reflection in computer generated images." *Communications of the ACM*, **19**(10):542–547, 1976.

[BSS93] Philippe Blasi, Bertrand Le Saëc, and Christophe Schlick. "A Rendering Algorithm for Discrete Volume Density Objects." *Computer Graphics Forum (Eurographics '93)*, **12**(3):201–210, 1993.

[CDN04] Florent Cohen, Philippe Decaudin, and Fabrice Neyret. "GPU-Based Lighting and Shadowing of Complex Natural Scenes." In *Siggraph'04 Conf. DVD-ROM (Poster)*, August 2004. Los Angeles, USA.

[CHH03] Nathan A. Carr, Jesse D. Hall, and John C. Hart. "GPU algorithms for radiosity and subsurface scattering." In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 51–59, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Coo84] Robert L. Cook. "Shade trees." In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 223–231, New York, NY, USA, 1984. ACM Press.

[DCS02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. “Interactive visualization of complex plant ecosystems.” In *Proceedings of the IEEE Visualization Conference*. IEEE, October 2002.

[DEJ99] Julie Dorsey, Alan Edelman, Henrik Wann Jensen, Justin Legakis, and Hans Køhling Pedersen. “Modeling and rendering of weathered stone.” In *SIGGRAPH ’99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 225–234, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[DGN99] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. “Reflectance and texture of real-world surfaces.” *ACM Trans. Graph.*, **18**(1):1–34, 1999.

[Dis98] Jean-Michel Dischler. “Efficient Rendering Macro Geometric Surface Structures With Bi-Directional Texture Functions.” In George Drettakis and Nelson Max, editors, *Rendering Techniques ’98*, Eurographics, pp. 169–180. Springer-Verlag Wien New York, 1998.

[DJ05] Craig Donner and Henrik Wann Jensen. “Light diffusion in multi-layered translucent materials.” In *SIGGRAPH ’05: ACM SIGGRAPH 2005 Papers*, pp. 1032–1039, New York, NY, USA, 2005. ACM Press.

[DN04] Philippe Decaudin and Fabrice Neyret. “Rendering Forest Scenes in Real-Time.” In *Rendering Techniques ’04 (Eurographics Symposium on Rendering)*, pp. 93–102, June 2004.

[DS03] Carsten Dachsbacher and Marc Stamminger. “Translucent shadow maps.” In *EGRW ’03: Proceedings of the 14th Eurographics workshop on Rendering*, pp. 197–201, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[DS06] Carsten Dachsbacher and Marc Stamminger. “Splatting indirect illumination.” In *SI3D ’06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 93–100, New York, NY, USA, 2006. ACM Press.

[EAJ05] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. “Interactive Rendering of Caustics using Interpolated Warped Volumes.” In *Graphics Interface*, May 2005.

[Eve01] C. Everitt. “Interactive order-independent transparency.” In *Technical report, NVIDIA Corporation, May 2001. Available at http://www.nvidia.com/*, 2001.

[FPB06] Guillaume Francois, Sumanta Pattanaik, Kadi Bouatouch, and Gaspard Breton. "Subsurface Texture Mapping." *IEEE Computer Graphics and Applications (to appear)*, 2006.

[GWS04] Johannes Günther, Ingo Wald, and Philipp Slusallek. "Realtime Caustics using Distributed Photon Mapping." In *Eurographics Symposium on Rendering*, pp. 111–122, 2004.

[Har96] J. C. Hart. "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces." *The Visual Computer*, **12**(10):527–545, 1996.

[HEG04] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. "Hardware accelerated per-pixel displacement mapping." In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pp. 153–158, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[Hei98] Wolfgang Heidrich. "View-Independent Environment Maps." In *Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware '98*, 1998.

[HH84] Paul S. Heckbert and Pat Hanrahan. "Beam tracing polygonal objects." In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pp. 119–127, New York, NY, USA, 1984. ACM Press.

[HK93] Pat Hanrahan and Wolfgang Krueger. "Reflection from layered surfaces due to subsurface scattering." In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 165–174, New York, NY, USA, 1993. ACM Press.

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. "Realistic, hardware-accelerated shading and lighting." In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 171–178, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[HV04] Xuejun Hao and Amitabh Varshney. "Real-time rendering of translucent meshes." *ACM Trans. Graph.*, **23**(2):120–142, 2004.

[IDN02] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. "An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware." *Computer Graphics Forum*, **21**(4):701–711, 2002.

[IDN03] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. "A Fast Rendering Method for Refractive and Reflective Caustics Due to Water Surfaces." In *Eurographics*, pp. 283–291, 2003.

[Jak00] Aleks Jakulin. "Interactive Vegetation Rendering with Slicing and Blending." *Eurographics (short presentations)*, 2000.

[JB02] Henrik Wann Jensen and Juan Buhler. "A rapid hierarchical rendering technique for translucent materials." In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 576–581, New York, NY, USA, 2002. ACM Press.

[JC98a] Henrik Wann Jensen and Per H. Christensen. "Efficient simulation of light transport in scences with participating media using photon maps." In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 311–320, New York, NY, USA, 1998. ACM Press.

[JC98b] Henrik Wann Jensen and Per H. Christensen. "Efficient Simulation of Light Transport in Scenes with Participating Media using Photon Maps." In *Computer Graphics (SIGGRAPH 98)*, pp. 311–320. ACM Press, 1998.

[Jen96] Henrik Wann Jensen. "Global Illumination Using Photon Maps." In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pp. 21–30, New York, NY, 1996. Springer-Verlag/Wien.

[JML01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. "A practical model for subsurface light transport." In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 511–518, New York, NY, USA, 2001. ACM Press.

[KR05] A Kolb and C Rezk-Salama. "Efficient Empty Space Skipping for Per-Pixel Displacement Mapping." In *Proc. Vision, Modeling and Visualization*, 2005.

[KS01] Jan Kautz and Hans-Peter Seidel. "Hardware accelerated displacement mapping for image based rendering." In *GRIN'01: No description on Graphics interface 2001*, pp. 61–70, Toronto, Ont., Canada, Canada, 2001. Canadian Information Processing Society.

[KTI01] T Kaneko, T Takahei, M Inami, N Kawakami, Y Yanagida, and T Maeda. "Detailed shape representation with parallax mapping." In *Proceedings of the ICAT 2001*, pp. 205–208, 2001.

[LGB02] H. Lensch, M. Goesele, P. Bekaert, J. Kautz, M. Magnor, J. Lang, and H. Seidel. "Interactive rendering of translucent objects." In *H. P. A. Lensch, M. Goesele, P. Bekaert, J. Kautz, M. A. Magnor, J. Lang, and H.-P. Seidel. Interactive rendering of translucent objects. In Proc. Pacific Graphics 2002, pages 214–224, Oct. 2002.*, 2002.

[Max98] Nelson Max. "Horizon mapping: shadows for bump-mapped surfaces." In *The Visual Computer 4, 2*, pp. 109–117, 1998.

[McG05] M McGuire. "Steep Parallax Mapping." In *I3D 2005 Poster*, 2005.

[MCT05] Wan-Chun Ma, Sung-Hsiang Chao, Yu-Ting Tseng, Yung-Yu Chuang, Chun-Fa Chang, Bing-Yu Chen, and Ming Ouhyoung. "Level-of-detail representation of bidirectional texture functions for real-time rendering." In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 187–194, New York, NY, USA, 2005. ACM Press.

[MKB03a] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth, and Hans-Peter Seidel. "Efficient Rendering of Local Subsurface Scattering." In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, p. 51, Washington, DC, USA, 2003. IEEE Computer Society.

[MKB03b] Tom Mertens, Jan Kautz, Philippe Bekaert, Hans-Peter Seidelz, and Frank Van Reeth. "Interactive rendering of translucent deformable objects." In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, pp. 130–140, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[MKB05] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth, and Hans-Peter Seidelz. "Efficient Rendering of Local Subsurface Scattering." *Computer Graphics Forum*, **24**:41–49, 2005.

[MN98] A Meyer and F Neyret. "Interactive volumetric textures." In *Eurographics Rendering Workshop*, pp. 157–168, 1998.

[MNP01] Alexandre Meyer, Fabrice Neyret, and Pierre Poulin. "Interactive Rendering of Trees with Shading and Shadows." In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pp. 183–196, London, UK, 2001. Springer-Verlag.

[NN94] Tomoyuki Nishita and Eihachiro Nakamae. "Method of displaying optical effects within water using accumulation buffer." In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 373–379, New York, NY, USA, 1994. ACM Press.

[NRH03] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. "All-frequency shadows using non-linear wavelet lighting approximation." *ACM Trans. Graph.*, **22**(3):376–381, 2003.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. "Relief texture mapping." In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 359–368, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[PC01] Frank Perbet and Maric-Paule Cani. "Animating prairies in real-time." In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 103–110, New York, NY, USA, 2001. ACM Press.

[PDC03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. "Photon Mapping on Programmable Graphics Hardware." In *Graphics Hardware*, pp. 41–50. Eurographics Association, 2003.

[Pel04] Kurt Pelzer. "Rendering Countless Blades of Waving Grass." *GPU Gems*, pp. 107–121, March 2004.

[PFT02a] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. "Root finding and non-linear sets of equations." In *Numerical Recipes in C*, pp. 354–360, 2002.

[PFT02b] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. "Root Finding and Nonlinear Sets of Equations." In *Numerical Recipes in C*, pp. 354–360. 2002.

[PHL91] John W. Patterson, Stuart G. Hoggar, and J. R. Logie. "Inverse Displacement Mapping." *Comput. Graph. Forum*, **10**(2):129–139, 1991.

[POC05] Fabio Policarpo, Manuel M. Oliveira, and Joao L. D. Comba. "Real-time relief mapping on arbitrary polygonal surfaces." In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 155–162, New York, NY, USA, 2005. ACM Press.

[Ree83] W. T. Reeves. "Particle Systems - a Technique for Modeling a Class of Fuzzy Objects." *ACM Trans. Graph.*, **2**(2):91–108, 1983.

[RSP07] Eric Risser, Musawir Shah, and Sumanta Pattanaik. "Faster Relief Mapping using the Secant Method." *journal of graphics tools*, **12**(3):17–24, 2007.

[RT87] Holly E. Rushmeier and Kenneth E. Torrance. "The zonal method for calculating light intensities in the presence of a participating medium." In *SIGGRAPH '87: Proceed-*

ings of the 14th annual conference on Computer graphics and interactive techniques, pp. 293–302, New York, NY, USA, 1987. ACM Press.

[SAL05] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. "Approximate Ray-Tracing on the GPU with Distance Impostors." In *Proceedings of Eurographics 2005*, 2005.

[SC00] P. Sloan and M. Cohen. "Interactive Horizon Mapping.", 2000.

[SHH03] Peter-Pike Sloan, Jesse Hall, John Hart, and John Snyder. "Clustered principal components for precomputed radiance transfer." *ACM Trans. Graph.*, **22**(3):382–391, 2003.

[SKP05] Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. "Real-time rendering of realistic-looking grass." In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pp. 77–82, New York, NY, USA, 2005. ACM.

[SKP07a] Musawir Shah, Jaakko Konttinen, and Sumanta Pattanaik. "Caustics Mapping: An Image-space Technique for Real-Time Caustics." *IEEE Transactions on Visualization and Computer Graphics, Vol. 13, No. 2*, pp. 272–280, 2007.

[SKP07b] Musawir Shah, Jaakko Konttinen, and Sumanta Pattanaik. "Image-Space Subsurface Scattering For Interactive Rendering Of Deformable Translucent Objects." *Accepted for publication in IEEE Computer Graphics and Applications*, pp. 272–280, 2007.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments." In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 527–536, New York, NY, USA, 2002. ACM Press.

[SLS05] Peter-Pike Sloan, Ben Luna, and John Snyder. "Local, deformable precomputed radiance transfer." In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 1216–1224, New York, NY, USA, 2005. ACM Press.

[SP99] G Schaufler and M Priglinger. "Horizon mapping: shadows for bump-mapped surfaces." In *Efficient displacement mapping by image warping*, pp. 175–186, 1999.

[SSK03] Mirko Sattler, Ralf Sarlette, and Reinhard Klein. "Efficient and realistic visualization of cloth." In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, pp. 167–177, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[SSS00] Brian E. Smits, Peter Shirley, and Michael M. Stark. "Direct Ray Tracing of Displacement Mapped Triangles." In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pp. 307–318, London, UK, 2000. Springer-Verlag.

[Sta95] Jos Stam. "Multiple Scattering as a Diffusion Process." In P. M. Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering)*, pp. 41–50, New York, NY, 1995. Springer-Verlag.

[Sta96] Jos Stam. "Random caustics: natural textures and wave theory revisited." In *SIGGRAPH '96: ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96*, p. 150, New York, NY, USA, 1996. ACM Press.

[SVL03] F. Suykens, K. Vom, A. Lagae, and P. Dutr. "Interactive rendering with bidirectional texture functions." *Computer Graphics Fourm 22*, September 2003.

[TS00] C. Trendall and A. Stewart. "General calculations using graphics hardware with applications to interactive caustics." In *Eurographics Workshop on Rendering*, pp. 287–298, 2000.

[VT04] M. Alex O. Vasilescu and Demetri Terzopoulos. "TensorTextures: multilinear image-based rendering." *ACM Trans. Graph.*, **23**(3):336–342, 2004.

[Wal03] Walsh. "Parallax Mapping with offset limiting." In *Infiniscape Tech Report*, 2003.

[Wat90] Mark Watt. "Light-water interaction using backward beam tracing." In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 377–385, New York, NY, USA, 1990. ACM Press.

[WD06] Chris Wyman and Scott Davis. "Interactive image-space techniques for approximating caustics." In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 153–160, New York, NY, USA, 2006. ACM Press.

[WHS04] Chris Wyman, Charles D. Hansen, and Peter Shirley. "Interactive Caustics Using Local Precomputed Irradiance." In *Pacific Conference on Computer Graphics and Applications*, pp. 143–151, 2004.

[WP95] Jason Weber and Joseph Penn. "Creation and rendering of realistic trees." In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 119–128, New York, NY, USA, 1995. ACM Press.

[WS03] M. Wand and W. Straßer. "Real-Time Caustics." *Computer Graphics Forum*, **22**(3):611–611, 2003.

[WTL03] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y Shum. “Generalized displacement maps.” In *Eurographics Symposium on Rendering*, pp. 227–233, 2003.

[WTL05] Rui Wang, John Tran, and David Luebke. “All-frequency interactive relighting of translucent objects with single and multiple scattering.” In *SIGGRAPH ’05: ACM SIGGRAPH 2005 Papers*, pp. 1202–1207, New York, NY, USA, 2005. ACM Press.

[WWT03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. “View-dependent displacement mapping.” *ACM Trans. Graph.*, **22**(3):334–339, 2003.

[Wym05] Chris Wyman. “An approximate image-space approach for interactive refraction.” In *SIGGRAPH ’05: ACM SIGGRAPH 2005 Papers*, pp. 1050–1053, New York, NY, USA, 2005. ACM Press.