INVERSE RENDERING METHODS FOR HARDWARE-ACCELERATED DISPLAY OF PARAMETERIZED IMAGE SPACES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ziyad Sami Hakura

October 2001

Abstract

One of the central problems in computer graphics is real-time rendering of physically illuminated, dynamic environments. Though the computation needed is beyond current capability, specialized graphics hardware that renders texture-mapped polygons continues to get cheaper and faster. We exploit this hardware to decompress “animations” computed offline using a photorealistic image renderer. The decoded imagery retains the full gamut of stochastic ray tracing effects, including indirect lighting with reflections, refractions, and shadows.

Rather than 1D time, our animations are parameterized by two or more arbitrary variables representing viewpoint positions, lighting changes and object motions. To best match the graphics hardware rendering to the input ray-traced imagery, we describe a novel method to infer parameterized texture maps for each object by modeling the hardware as a linear system and then performing least-squares optimization. The parameterized textures are compressed as a multidimensional Laplacian pyramid on fixed size blocks of parameter space. This scheme captures the coherence in animations and, unlike previous work, decodes directly into texture maps that load into hardware with a few, simple image operations. High-quality results are demonstrated at compression ratios up to 800:1 with interactive playback on current consumer graphics cards.

To enable plausible movement away from and between the pre-rendered viewpoint samples, we extend the idea of parametric textures to parametric environment maps. Segmenting the environment into layers, and picking simple environmental geometry that closely matches the actual geometry of the environment better approximates how reflections move as the view changes. Unlike traditional environment maps, we achieve local effects like self-reflections and parallax in the reflected imagery.

Finally, we introduce hybrid rendering, a scheme that dynamically ray traces the local geometry of refractive objects, but approximates more distant geometry by layered, parameterized environment maps. To limit computation, we use a greedy ray path shading model that prunes the binary ray tree generated by refractive objects to form just two ray paths. We also restrict ray queries to triangle vertices, but perform adaptive tessellation to shoot additional rays where neighboring ray paths differ sufficiently. We demonstrate highly specular glass objects at a significantly lower and more predictable cost than ray-tracing, and anticipate future support for local ray-tracing in graphics hardware will make this approach ideal for real-time rendering of realistic reflective and refractive objects.

Acknowledgments

I would like to thank the many people who have helped make this dissertation possible.

I would first like to thank Anoop Gupta for his gracious mentorship throughout my graduate studies. I am especially thankful to Anoop for providing me with numerous opportunities to develop and broaden my research skills. He made it possible for me to take part in the Stanford FLASH Multiprocessor project where I was exposed to systems research. He also inspired me to apply my architectural interests to computer graphics in the FLASH Graphics project. In addition, he provided me with an opportunity to visit Microsoft Research for two years in the middle of my graduate studies where he helped focus my energy on rendering algorithms research for my dissertation. I owe Anoop a great deal for making sure that I was constantly challenged and aspiring to do important research.

I would next like to thank John Snyder who has closely advised me throughout the time that I have worked on my dissertation. I am greatly indebted to John for his dedication to our work, and his invaluable advice and assistance. Working with John has been a very rewarding experience. I especially value his insights, his keen eye for producing high quality computer graphics renderings, and his emphasis on rigor in research. John's continuous encouragement and support has helped make my experience more enjoyable, and the friendship I have developed with him is one that I will always cherish.

Pat Hanrahan has kindly served on both my orals committee and my reading committee. I would like to thank him for thoughtful discussions and for his valuable feedback along the way.

I would like to thank Jed Lengyel for the time and assistance that he has given me. It was in joint conversations with Jed Lengyel and John Snyder that I became inspired to work on the topic of this dissertation.

During my two year stay at Microsoft Research and in later visits, I benefited greatly from the feedback I received at my graphics lunch talks. In particular, I thank Turner Whitted, Michael Cohen, Peter-Pike Sloan, Brian Guenter, Hugues Hoppe, Jim Blinn, David Salesin, Kirk Olynyk, Chuck Jacobs, Richard Szeliski, Charles Loop, and Zicheng Liu for their valuable discussions and comments. I would like to especially thank Turner for his support and encouragement, and for proofreading our papers. I would also like to thank Peter-Pike for his helpful critique of my orals presentation. Extremely useful advice and discussions were provided by Henrique Malvar, Philip Chou and Brian Guenter on compression, John Platt on optimization and regularization, and Steve Marschner on solving for texture maps and ray-tracing. I enjoyed many informal conversations with Mike Sinclair and Victor Bahl and the friendship of members of the Collaboration and Multimedia Systems group at Microsoft Research.

I have learned from discussions with numerous colleagues and friends in the graphics group at Stanford, including Ravi Ramamoorthi, Li-Yi Wei, Sean Anderson, David Koller, Robert Bosch, Gordon Stoll, Homan Igehy, Matthew Eldridge, Milton Chen, John Owens, Olaf Hall-Holt, Maneesh Agrawala, Szymon Rusinkiewicz, Henrik Wann Jensen, Bill Mark, Bennett Wilburn, Marcus Magnor, Chris Stolte, Matt Pharr, Philippe Lacroute, Matthew Regan, and Benjamin Zhu. I especially thank Ravi for many illuminating discussions and helpful critique on both my paper and orals presentations. I thank Sean and Steve for helpful critique on my orals presentation, and for answering my occasional graphics questions. I would like to thank both Ravi and Li-Yi for help with video preparation and printing at paper deadlines. I have enjoyed working with Homan, Matthew, Milton, Gordon, and John in the Stanford FLASH Graphics project. I also thank Professor Marc Levoy for useful comments at my graphics lunch talks at Stanford, and for providing me with visualizations of light field sample accesses. Peter Shirley's Eon ray tracer was a valuable research tool.

I have also had the pleasure of working with members of the FLASH team in my early years at Stanford and from whom I learned much about building systems, including David Ofelt, Jeff Kuskin, Ravi Soundararajan, Joel Baxter, Jules Bergmann, Jeff Gibson, John Heinlein, Mark Heinrich, Richard Ho, Hema Kapadia, Shankar Govindaraju, and Dave Nakahira. I also enjoyed the friendship of members of the SimOS team at Stanford, whose offices were close to mine, including Scott Devine, John Chapin, Kinshuk Govil, Ben Verghese, and Dan Teodosiu. I am also thankful to Professors Kunle Olukotun, Mark Horowitz, and Mendel Rosenblum for their friendship throughout my time at Stanford.

I am thankful to Charlie Orgish, John Gerth, Thoi Nguyen, and Kevin Colton for their hard work in keeping the computers up and running smoothly, and their friendship throughout my graduate school experience. I am also thankful to Darlene Hadding, Ada Glucksman, and Heather Gentner for dealing with administrative matters and effectively hiding the bureaucratic aspects of Stanford from me. I am especially thankful to Darlene for being extremely helpful in the latter years of my graduate studies. The work that they did made my life at Stanford much easier and is greatly appreciated.

I would like to thank some very good friends who made my life in graduate school enjoyable: Ulrich Stern, Ravi Soundararajan, Johannes Helander, Sean Anderson, Ravi Ramamoorthi, Kanna

Shimizu, Rebecca Yang, Luke Chang and David Koller. I am grateful to Ravi Soundararajan and Kanna for being good officemates and great friends. I am especially thankful to Ulrich and Rebecca for our many dinner and movie excursions. I am grateful to Johannes for his close friendship during my stay at Microsoft and the many exhilarating biking and hiking experiences we embarked on in the beautiful wilderness of Washington State. I am also thankful to other people I came across at Stanford and Microsoft, including Rebecca Xiong, Ramesh Raskar, Rachid Helaihel, Wael Noureddine, David Park, Navin Chaddah, and numerous others I may have inadvertently forgotten, for their friendship as I plodded along towards the Ph.D.

Finally, I would like to thank my family – my father Sami, mother Rabab, and sisters Dima and Dalia – for their unfailing love and support. I would also like to thank my sister's husband Geert Almekinders and their charming daughter Sophie who have become important people in my life. I am deeply grateful to my father who instilled in me the desire to achieve great things, provided the support that helped me persevere through graduate school, and whose wish it was to see me complete my dissertation. I owe him everything.

To my parents,
Sami and Rabab

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer graphics is about generating an image from an abstract description of a world which can be either real or imaginary. Perhaps one of the more fascinating applications of computers is in rendering images that are sufficiently realistic so as to be indistinguishable from reality. In addition to being visually appealing, the ability to generate such realistic images has applications that permeate all aspects of our lives. Common applications today are computer-aided design, realistic visualization of scientific phenomenon, photorealistic computer games, and computer-generated animated films. In the near future, we can imagine augmented reality applications in medicine and engineering, telepresence in virtual meetings, and user friendly three-dimensional human-computer interfaces in ubiquitous computing.

The central problem in computer graphics is real-time rendering of physically-illuminated, dynamic environments. Over the past few decades, researchers in computer graphics have made tremendous progress towards attaining this goal. Their efforts can be broadly categorized into two areas: *realistic image synthesis* and *interactive rendering.*

In the area of realistic image synthesis, the challenge has been how to accurately model the world, *modeling*, and how to accurately simulate the transport of light to generate an image, *rendering.* In addition, assuming the world is not static, there is the challenge of describing the motion of objects, *animation.* The quest for realism has increasingly lead to more accurate simulation of physical processes that are visually perceptible. Unfortunately, these methods are by their very nature highly computationally intensive and consequently, are so far only useful for offline rendering. Popular animated films today, for example Toy Story, use such photorealistic rendering, and require several hours per frame.

In contrast, researchers in interactive rendering have been primarily concerned with producing the best possible images within a fixed time interval. This interval is governed by the ability of the human visual system to perceive continuous motion in discrete frames, the *fusion rate*, which is around 20 Hz. The stringent timing requirements of interactive graphics applications has constrained the choice of rendering algorithms to what can be feasibly implemented in hardware. Though this hardware, generally referred to as the *graphics pipeline*, has increased in performance and functionality over several generations, it is still relatively crude in its ability to render realistic images compared with software algorithms used in offline rendering.

In this work, we attempt to bridge the gap between realistic image synthesis and interactive rendering. Though the computation needed to perform highly realistic interactive rendering is beyond the current capability, traditional graphics hardware continues to get cheaper and faster. We exploit this hardware to efficiently compress offline rendered multi-dimensional parameterized “animations”. These animations can be decoded interactively using graphics hardware and exhibit the full gamut of realistic image synthesis effects, such as indirect lighting with reflections, refractions, and shadows.

1.1 Parameterized Image Spaces

Rather than simply being one dimensional in time, our animations are parameterized by two or more variables representing viewpoint positions, lighting changes and object motions. Figure 1.1 illustrates an example of a 2D parameterized image space combining viewpoint movement along a 1D trajectory with independent 1D rotation of a ring object. The individual images in this figure are rendered by *ray-tracing*, presently the most popular method of generating realistic images. Note the accurate lighting, realistic local reflections, refractions and soft shadows in each image. This quality of rendering cannot be attained in real-time using present-day graphics hardware.

The main objective in this work is to give the user the ability to interactively navigate the parameterized image spaces. This means the user should be able to move through the space in an arbitrary direction and simultaneously along more than one dimension. This is in contrast to watching computer-generated animated films where there is a single dimension and where we cycle through the images in a fixed order. Below, we present three example applications of parameterized image spaces.

- **Visualizing Complex Machinery**
  One possible application of parameterized image spaces is in visualizing complex machinery. In this space, we would like to be able to watch the machinery animate with time. In addition, we would like to be able to watch this animation from any point along a 1D circle that surrounds the machinery or, perhaps, from any point on the 2D hemisphere that encloses it. The ability to see the animation from any point in the 2 or 3-dimensional space enables us to better understand how the machinery works.

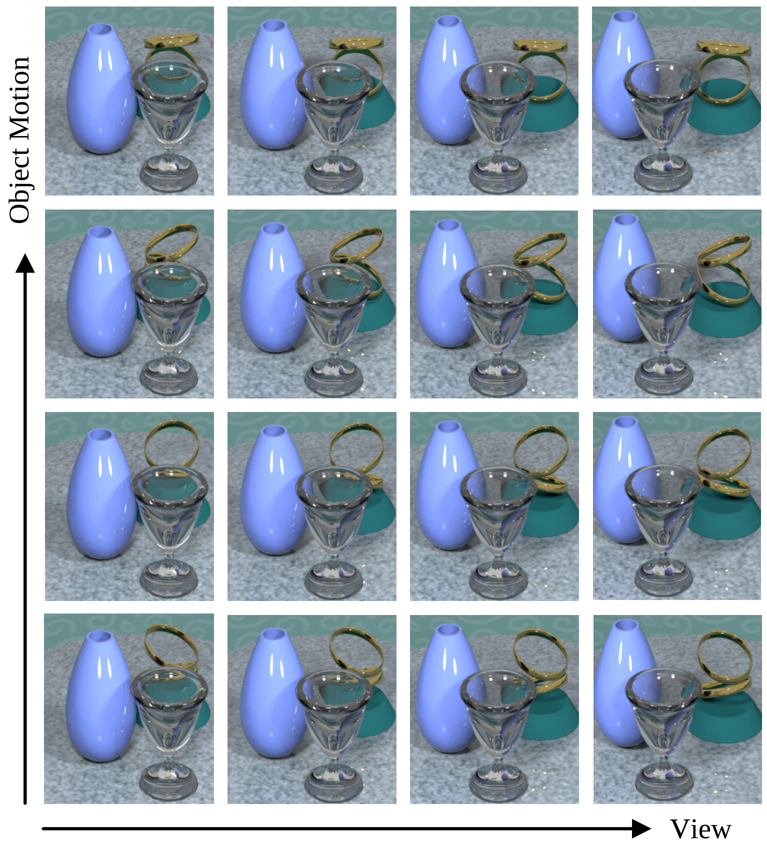


Figure 1.1: **Example of a 2D parameterized image space.** This space combines viewpoint movement along a 1D trajectory with independent 1D rotation of the ring object.

- **Realistic Cockpit Rendering in a Flight Simulator**
  Another example application is in producing a realistic cockpit rendering in a flight simulator. In this space, the possible parameters are head position, the discrete positions of the switches and knobs in the cockpit, and the time of day. Accounting for head position allows for view dependent effects such as glossy and specular reflections in the interior of the cockpit. The switches and knobs in the cockpit control the displays and LEDs, and thus effect the lighting emitted in the interior. The time of day effects the amount of outdoor lighting transported through the cockpit windows. Taken together, these parameters allows for a high level of image realism in the interior of a cockpit.

- **Interactive Animated Film**
  A third example application is in making an interactive version of an animated film, like Toy Story from Pixar Animation Studios. In this example, one parameter can be limited viewpoint

motion. In a virtual reality setting, the ability to see parallax corresponding to head motion gives a much more immersive effect. By tracking head motion using a head-mounted display, and allowing for limited head movements near the original trajectory, we can give the person watching the film the sense of actually being “in the scene”.

Another set of possible parameters are character parameters. Recently, there has been much work in character animation to realize some very rich animations with a minimal number of user-controlled parameters [Ros98, Gle98, Pop99]. We can imagine having parameters that control the emotional states of leading characters in an animated film. For example, by controlling the happiness or sadness of Woody in Toy Story, we can effect how Woody walks and his facial expressions.

As we have just shown, there are many possible parameters. It is the responsibility of the content author to decide the dimensionality of the image space, what parameters the dimensions represent, and how densely to sample the space.

Before describing our system architecture for interactively displaying parameterized image spaces, we briefly describe the algorithms that have been developed for realistic image synthesis, and the graphics pipeline implemented for interactive rendering in the next two sections.

1.2 Realistic Image Synthesis

The emergence of raster graphics in the late 1960s, where each point on a display screen, or *pixel*, is represented by a color value made it possible to portray the surface appearance of three-dimensional objects. The first models for shading surfaces were developed by Bouknight [Bou70], Gouraud [Gou71], and Phong [Pho75]. These models were *ad hoc*, in that they were not developed based on physical principles. The models were also *local*, in that each surface was shaded without considering any other surfaces in the environment. Thus, the images generated did not account for reflection of light between surfaces, and shadows caused by the obstruction of light sources were missing.

The first algorithm that took *global* illumination phenomena into account was developed by Whitted [Whi80]. This algorithm, known as *ray-tracing*, worked by recursively tracing rays through an environment starting with rays emanating from the eye-point. Thus, it could account for indirect lighting in the form of reflections, refractions and shadows. There have been many improvements to the basic ray tracing algorithm [Gla89]. Cook et al. introduced stochastic ray tracing [Coo84a] to handle glossy reflections and soft shadows, Ward et al [War88] introduced a method for incorporating diffuse interreflection between surfaces, and Shirley [Arv86] introduced the use of backwards ray-tracing to render caustics. Recent methods for increasing the robustness and efficiency of these techniques include the work of Shirley [Shi95], Jensen [Jen96], and Veach [Vea97].

While ray-tracing simulated the transport of light in an environment, models were needed to describe the interaction of light with the surfaces that the rays intersected. Inspired by work from the fields of radiative heat transfer and illumination engineering, Blinn [Bli77] and Cook and Torrance [Coo82] introduced the first local reflection models that were physically based. Kajiya [Kaj85] generalized the Cook-Torrance model to handle anisotropic surfaces such as brushed metals, cloth and hair. Models to handle furry materials were also introduced [Kaj89, Mil88]. More recently, models that account for subsurface scattering in translucent materials, such as marble, skin, and milk, have been developed [Han93, Sta95, Dor99, Pha00, Jen01].

In the mid 1980s, radiosity methods from the field of radiative heat transfer began to be applied to realistic image synthesis [Gor84]. These view-independent methods were used to solve for the interreflection of light in an environment consisting of ideal (Lambertian) diffuse surfaces. As with ray-tracing, there have been many improvements to the basic algorithm [Coh93]. While radiosity methods were later extended for glossy and specular surfaces [Imm86], they are typically not used to simulate directional lighting effects, and are instead combined with ray-tracing solutions [Wal87, Sil89, Shi90, Che91].

The early rendering algorithms modeled light as point sources or as distant directional sources. Verbeck and Greenberg [Ver84] observed that accurate lighting requires modeling the geometry of the light source and the intensity distribution as a function of wavelength and direction. Accounting for the geometry of a light source enabled them to create the effect of penumbra on partially shadowed geometry. Modeling the intensity distribution made possible light pattern effects typically seen in real environments.

Kajiya [Kaj86] introduced the *rendering equation* in 1986 which tied together the illumination models. Kajiya accounted for the transport of light in an environment through an integral that considered the incoming light distribution, the surface bidirectional reflectance function, the emitted light, and the scattering of light through an intervening material.

As shading models and lighting models became more complex, it became evident that shading systems that uniformly evaluated a single parameterized shading expression for all surfaces were both inadequate and inefficient. Cook [Coo84b] introduced shade trees that allowed flexible tree-structured expressions for surface reflectance models, light source models, and atmospheric effects. Perlin [Per85] took the idea further by allowing the shading expressions to be specified using a full programming language. Hanrahan and Lawson [Ren89, Han90] brought together features of Cook and Perlin's systems in an abstract shading language called *RenderMan*, presently the standard language for writing shading expressions. A major feature of this language is that it is independent of any specific illumination models, software algorithms or hardware implementations. It has been found that because surface reflectance models are often phenomenologically based, many materials are more easily defined with procedurally defined shaders than with mathematical formulae.

In conclusion, much progress has been made towards offline rendering of realistic images. In this work, we seek to exploit these methods to interactively render parameterized image spaces at a comparable level of realism.

1.3 Interactive Rendering

In parallel to the development of realistic image synthesis techniques is the development of interactive rendering systems. Interactive rendering systems generally strive to produce images with the highest degree of image complexity and realism at interactive rates. In pursuit of this goal, systems that take advantage of specialized hardware have been built. Architects of these systems have grappled with several challenging questions. These include what rendering primitives to support? how to effectively exploit parallelism? how to design memory systems to maximize throughput? and where to draw the line between specialized hardware computation and general purpose computation?

The graphics systems that have been built can be classified by generations, Akeley [Ake93], where each generation is characterized by the target rendering capabilities for which performance is maximized. The first generation of graphics hardware in late 1970s and early 1980s was primarily good at drawing wireframe models of 3D geometry. The implementation of deep framebuffers and hidden surface elimination in the second generation of hardware in the late 1980s made it possible to efficiently draw Gouraud shaded polygons. In the early 1990s, the third generation of machines with fast polygon texture mapping capabilities and anti-aliasing were introduced. We are presently in the fourth generation of graphics hardware, in an extension of the classification by Hanrahan [Han97], where hardware is particularly good at flexible lighting, shading and texturing of polygons.

One common characteristic of interactive rendering systems is that they employ a standard *graphics pipeline* for rendering, illustrated in Figure 1.3. The distinguishing feature of this pipeline is that it *projects* geometric primitives to the screen, rather than *tracing* rays from the eye-point into the image. We will make this distinction more concrete below. The functionality provided by the graphics pipeline is exposed to the programmer in the form of application programming interfaces (APIs). Two very similar APIs commonly used today are OpenGL (Open Graphics Library) [Seg92, Nei93] and Direct3D [DX8].

The image in Figure 1.4(b) is representative of what can be rendered on hardware that implements the graphics pipeline. To make clear the capabilities and limitations of this pipeline, we briefly explain how this image was generated. A more complete description of the graphics pipeline can be found in [Fol90].

As illustrated in Figure 1.3, the pipeline consists of four stages. In the first stage, the graphics

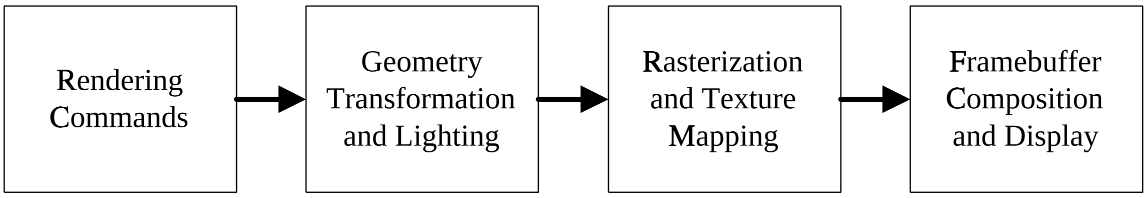

Rendering Commands
Geometry Transformation and Lighting
Rasterization and Texture Mapping
Framebuffer Composition and Display

Figure 1.2: **The graphics pipeline in interactive rendering systems.**

application performs rendering calls through the API to define the image to be rendered. In our example, calls are made to define the surface geometry of each object, the textures that are mapped onto each surface, the material properties of the surfaces, and the lighting in the environment.

The surface geometry of objects is typically defined in terms of triangles. In the second stage, the triangles are projected from their 3D positions in space to the 2D screen using a perspective mapping. Color values are also computed for each vertex based on the specific material parameters and general lighting conditions. In the example image, there are two lights in the environment which are modeled as point sources. All the vertices, except those for the cup, are lighted using a combination of a simple diffuse reflection model, computed for each light source independently, and a global ambient illumination constant. The specular highlights on the vase are added separately using a simplified version of the Phong specular reflection model supported in hardware. For the cup, which is modeled as a purely reflective object, environment map texture coordinates are computed per vertex by taking the direction of the reflection ray at each vertex and intersecting this with a finite cube whose bottom rests on the table and other sides extend to the walls in the environment.

In the third stage, the triangles are scan converted to identify the pixels that are in the interior of triangles. These pixels are shaded by interpolating the color values at the vertices using Gouraud interpolation. In addition, the shaded pixel colors are modulated with texture values by indexing into texture maps. The per vertex texture coordinates are interpolated to compute per pixel texture addresses. In our example image, static 2D texture images are mapped onto all objects, except for the vase and cup. In the case of the cup, the environment map texture coordinates at the vertices, dynamically computed in the previous stage, are interpolated and used to index into a static environment map.

The shadows seen in the image are also computed in the third stage by performing a separate rendering pass for each light source. The state of shadow of each pixel is determined by indexing into a precomputed shadow map, specific for a particular light source, and modulating the shadow result with the overall shaded and textured value. The shadow result depends on whether the surface point is the closest point directly visible to the light source.

In the last stage of the pipeline, a Z-buffer algorithm is used to eliminate hidden surfaces, and

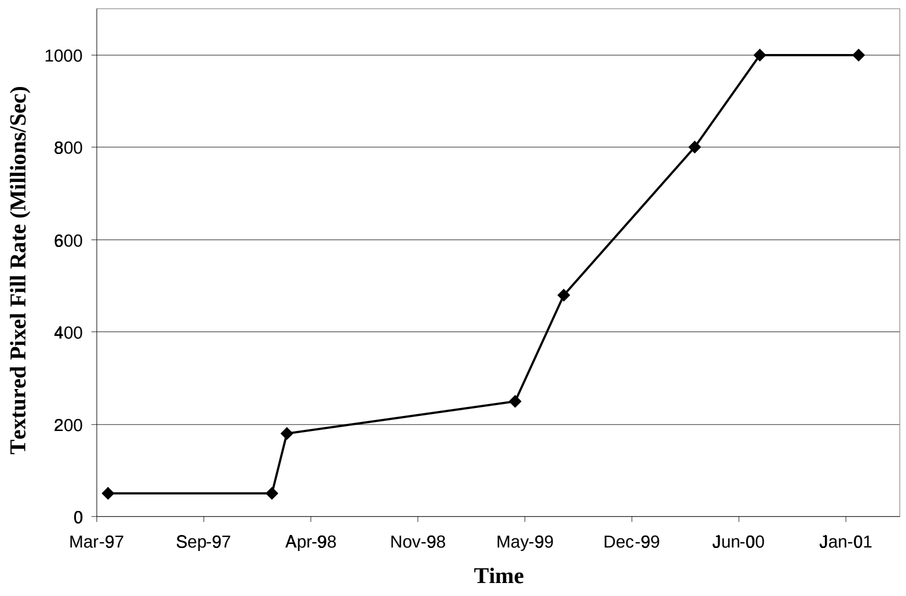
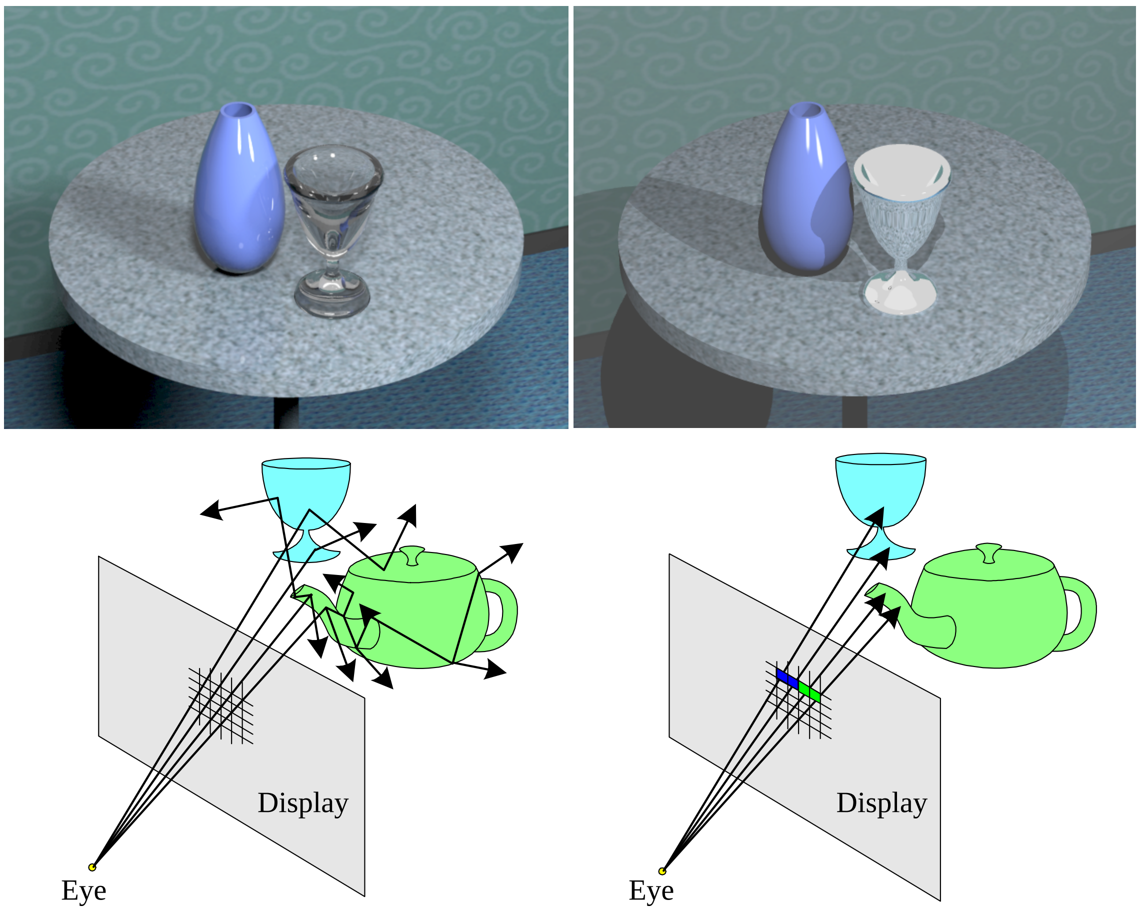


Figure 1.3: **Trend in textured pixel fill rate for commodity PC graphics cards.** Data obtained from [NV].

pixels that are found to be visible are composited with the framebuffer. The image is also anti-aliased by supersampling the image and filtering a neighborhood of pixels. Finally, the image stored in the framebuffer is gamma corrected and displayed on the screen.

In recent years, the graphics pipeline has seen drastic improvements in both absolute performance and cost/performance. Figure 1.3 shows the trend in textured pixel fill rate over the last four years for commodity PC graphics cards. As shown, there has been a twenty fold increase from 50 million to 1 billion textured mapped pixels per second. Other metrics of graphics performance, such as the triangle transformation rate have also been going up correspondingly. These improvements in performance and cost can be attributed to both better architectural implementations, such as the use of texture and vertex caches to reduce memory bandwidth requirements, and better semiconductor technology, enabling the use of higher clock frequencies, more transistors to exploit parallelism, and fewer components to build a complete system.

Figure 1.4 compares the graphics pipeline image with one produced using a ray-tracer. The ray-traced image is clearly more realistic. In the ray-traced image, we see accurate lighting, local reflections, refractions and soft shadows. In contrast, the local reflections between the vase, cup and table are missing in the image produced by the graphics pipeline, we are unable to simulate refractions in the glass cup, and the shadows have hard boundaries.



(a) Ray-Traced (b) Graphics Pipeline Hardware

Figure 1.4: **Comparison between ray-tracing and graphics pipeline rendering.**

The diagrams below the corresponding images help us better understand the fundamental difference between these two kinds of rendering. With ray-tracing, reflective and refractive objects map each incoming ray into a number of outgoing rays, according to a complex, spatially-varying set of multiple ray "bounces". In contrast, the graphics pipeline rasterizes geometry with respect to a constrained ray set – rays emanating from a point and passing through a uniformly parameterized rectangle in 3D. Thus, graphics hardware cannot be expected to accurately simulate non-local illumination effects. However, graphics hardware is well-suited for interactive rendering because the memory access patterns are regular and the computation is both low and predictable at each stage of the pipeline.

In terms of rendering speed, the computation needed to perform ray-tracing in real-time exceeds

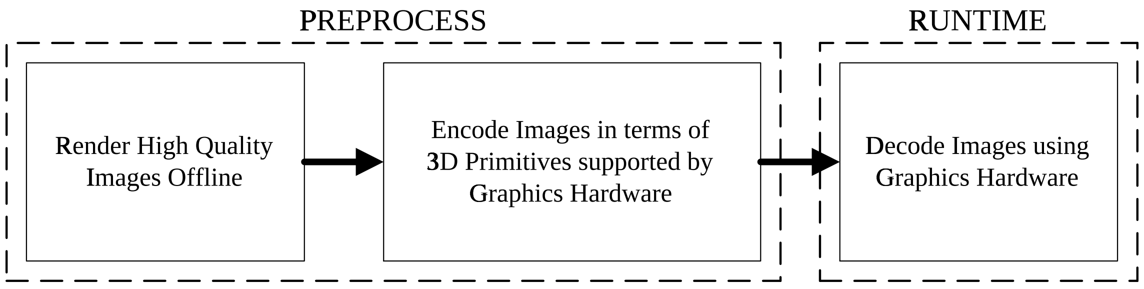

PREPROCESS
RUNTIME
Render High Quality Images Offline
Encode Images in terms of 3D Primitives supported by Graphics Hardware
Decode Images using Graphics Hardware

Figure 1.5: **Three Step System Model.**

the current capability of general purpose computers by at least four to five orders of magnitude, as it can presently take from tens of minutes to hours to render a single image. In comparison, commodity graphics hardware today is capable of rendering relatively complex scenes, containing more than one million triangles, at interactive rates.

1.4 System Architecture

We have shown that interactive rendering systems are optimized for graphics pipeline rendering. While this allows for fast rendering of texture-mapped geometry, it also means that graphics hardware alone cannot efficiently produce images with accurate global illumination effects. We, therefore, propose the following three step model, illustrated in Figure 1.5, for interactively rendering parameterized image spaces with realistic image quality.

The first step of the model consists of offline rendering of the multi-dimensional parameterized image space. The idea is to take advantage of the realistic image synthesis techniques that have been developed to produce images with high realism. In our work, we render images using a modified version of Eon, a Monte Carlo distribution ray-tracer [Coo84a, Shi92, Shi96]. Note, however, that our architecture supports any of the image synthesis techniques described earlier in Section 1.2.

In the second step, we compress the pre-rendered images. The goal is to achieve high compression, and at the same time allow for fast decoding at run-time. This goal is accomplished by encoding the images in terms of 3D primitives supported by graphics hardware, and in particular, in terms of 3D geometry and texture maps. This step is akin to compiling programs to run efficiently on a particular processor. In our case, we are compiling the parameterized image space to run efficiently and with a comparable level of image quality on a particular pipeline rendering model facilitated by graphics hardware.

Finally, in the third step, we treat the graphics hardware as a decoder, and take full advantage of its fast rendering capabilities to reconstruct the original high quality images. In addition to being able to accurately reproduce the original images, we would like our system to be able to interactively render plausible images between and away from the pre-rendered samples.

The first two steps of the model are performed offline as a preprocess, whereas the last step is performed interactively at runtime.

1.5 Research Contributions

There are four major contributions in this thesis, listed below.

- **Inverse rendering method for inferring texture maps.** As mentioned in Section 1.4, we compile the pre-rendered image spaces in the second step of our system architecture for efficient compression and fast decoding. In our system, compression and decoding are facilitated by converting the original image representation produced in the pre-rendering step into a texture representation for each object. Thus, an algorithm for *inferring* a texture for an object from a corresponding ray-traced image is needed. To best match the graphics pipeline rendering to the input ray-traced images, we have developed a novel method that models the pipeline as a linear system, and then performs least-squares minimization. We demonstrate a sharper match to the original ray-traced images when rendering with textures inferred using this inverse fitting method than with textures computed using either an alternative forward mapping approach or by ray-tracing directly into texture maps.

- **Hardware accelerated decoding of compressed parameterized image spaces.** Inferring a separate texture map for an object at each point in the image space results in a parameterized texture. We compress the parameterized textures for each object taking advantage of multi-dimensional coherence, and at the same time, allowing for fast real-time decoding on graphics hardware. We demonstrate a complete system that achieves high quality rendering of parameterized image spaces at compression ratios up to 800:1 with interactive playback on current consumer graphics cards.

- **Parameterized environment map representation for plausible movement away from pre-rendered samples.** We introduce parameterized environment maps as an alternative to parameterized texture maps, so that we can render plausible images when we move the viewpoint away from the pre-rendered image samples. By segmenting the environment into layers, and picking simple environmental geometry that closely match the actual geometry of the environment, we can better predict how reflections move as the view changes. Unlike with traditional environment maps, we achieve local effects like self-reflections and parallax in the reflected imagery.

- **Hybrid rendering for refractive objects.** We introduce hybrid rendering, a scheme that dynamically ray traces the local geometry of refractive objects, but approximates more distant geometry by layered, parameterized environment maps. To limit computation, we use a greedy ray path shading model that prunes the binary ray tree generated by refractive objects to form just two ray paths. We also restrict ray queries to triangle vertices, but perform adaptive tessellation to shoot additional rays where neighboring ray paths differ sufficiently. We demonstrate plausible movement away from pre-rendered samples for glass objects.

1.6 Thesis Organization

The next chapter discusses related work.

Chapter 3 describes our inverse rendering method for inferring texture maps which we use throughout this thesis, and quantitatively compares this method with an alternative forward mapping approach and with ray-tracing directly into texture maps.

Chapter 4 describes our multi-dimensional compression scheme for parameterized textures, and discusses the system-level issues in our runtime environment. We also present compression and playback performance results for two examples of parameterized image spaces.

Chapter 5 describes parameterized environment maps. As part of this description, we compare with traditional, or static, environment maps, as well as with light field methods that capture specular objects over an entire viewspace. We also discuss issues related to layering of environment maps and environment map inference. We present performance and image results for both highly reflective objects and glossy objects.

Chapter 6 describes our hybrid rendering algorithm, and presents performance and image results for highly refractive objects.

Finally, **Chapter 7** summarizes the major results from this research, presents recommendations for future graphics hardware, and discusses directions for future work.

Chapter 2

Related Work

There are four main areas of related work: *image-based rendering, hardware shading models, texture recovery/model matching,* and *compression.* In this chapter, we briefly discuss each of these areas, and compare with our approach of parameterizing texture and environment maps for interactive rendering of realistic images.

2.1 Image-Based Rendering (IBR)

Image-based rendering (IBR) relies upon photometric observations of an environment to construct images. This is in contrast to traditional geometry-based rendering, where an explicit description of objects in the environment is given either in the form of boundary regions between elements or a sampled volumetric description. IBR has sought increasingly accurate approximations of the plenoptic function [Ade91, McM95], or spherical radiance field parameterized by 3D position, time, and wavelength. The "plenoptic function" of Adelson and Bergen is a parameterized function that describes everything that is visible from a given point in space. IBR has proven particularly useful in situations where the geometry in the environment is not known, such as images of the real-world captured using a still or moving camera. Our work instead focuses on synthetic imagery where the geometry and material properties of objects in the environment is known. Another difference is that for synthetic scenes, the time and viewpoint parameters of the plenoptic function can be generalized. We are free to parameterize the radiance field based on time, position of lights or viewpoint, surface reflectance properties, object positions, or any other degrees of freedom in the scene, resulting in an arbitrary-dimensional parameterized animation.

Environment maps (EMs) [Bli76, Gre86], which store a sphere of radiance incident at a point, were originally developed for approximating reflections of an environment on an object surface. It was recognized early on that environment maps constructed from 360-degree panoramic images may also be used to display any outward looking direction from a fixed viewpoint giving the impression of standing inside an environment. The Apple QuickTime VR system [Che95] used this rendering approach, with multiple environment maps acquired at key locations within a scene. The user is able to navigate the environment by discretely "hopping" between locations, and at each location can look in an arbitrary direction.

Chen and Williams [Che93] pioneered the approach of interpolating between images from multiple viewpoints. Their system established pixel correspondence between any pair of images using depth information at each pixel. To generate in-between views, they simply interpolated along the direction of pixel flow. One of the challenges of this approach is filling in "holes", or disocclusions, that can result when regions in the environment that are occluded in all the source images become visible at intermediate views. This method also implicitly relies on diffuse surface reflectance since it potentially combines pixels from a variety of viewpoints to render a single image.

Levoy and Hanrahan [Lev96] and Gortler et al. [Gor96] reduced the plenoptic function to a 4D field, allowing view interpolation with view-dependent lighting. The *Light Field* approach of Levoy and Hanrahan [Lev96] captured the environment with a 2D array of 2D images acquired on a regularly sampled plane. In this scheme, the plenoptic function is parameterized by two parallel planes, one located at the camera plane and another at a "focal" plane. The radiance of any desired viewing ray can be computed by finding its (u, v) and (s, t) intersection coordinates with each of the two planes, and performing quadralinear interpolation in the neighborhood of (u, v, s, t) to avoid aliasing. In our terminology, the uv plane is the camera plane, and the st plane is the focal plane. The *Lumigraph* of Gortler et al. [Gor96] is a similar rendering algorithm. One of the contributions of the Lumigraph is that it can take advantage of depth information to more accurately reconstruct rays from a sparse sampling of source cameras. The depth information is used to adapt the (s, t) coordinates on the focal plane so that the neighborhood of rays indexed from the 4D field intersects the same geometric location as the desired viewing ray. The depth information is given as approximate geometric models of objects in the environment.

Reconstructing an image from a particular view with either the light field or lumigraph approaches may require visiting an irregular scattering of samples over the entire 4D field. Figure 2.1 illustrates the samples in an example light field that are accessed from two different viewpoints. The accesses are especially scattered among multiple source images when the desired viewpoint is near the focal plane, as shown in Figure 2.1(b). Accessing samples from multiple source images reduces memory coherence. In addition, to achieve fast decoding at run-time, compression algorithms must support fast random access into the 4D field. Typically, support for fast random access comes at the cost of the amount of signal coherence that is exploited, thus reducing the overall compression efficiency.

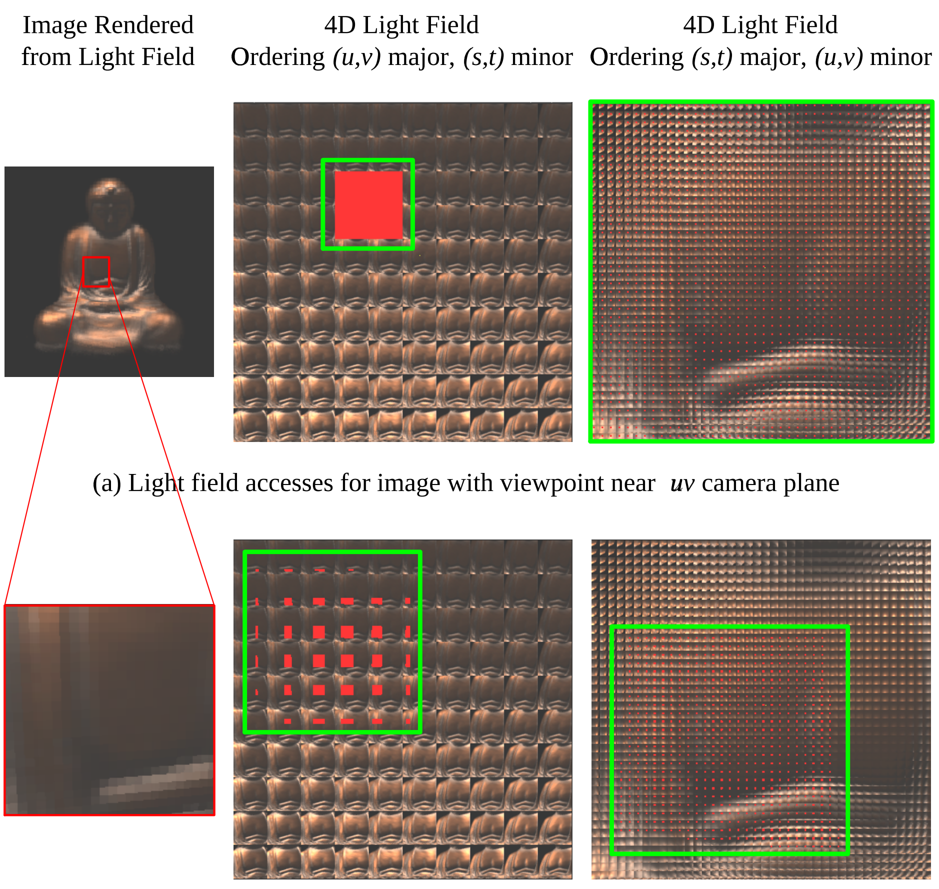


Image Rendered from Light Field
4D Light Field Ordering (u,v) major, (s,t) minor
4D Light Field Ordering (s,t) major, (u,v) minor
(a) Light field accesses for image with viewpoint near uv camera plane
(b) Light field accesses for closeup image with viewpoint near st focal plane

Figure 2.1: **Visualization of Light Field sample accesses.** The uv plane is the camera plane, and the st plane is the focal plane. The images on the far left are screen images rendered from the light field. The images in the middle represent the source images that make up the light field ordered by their corresponding (u, v) coordinates on the camera plane. The images on the far right represent the source images that make up the light field with a different ordering of samples where the major coordinates are (s, t) coordinates, and within each block, the minor coordinates are (u, v). In both the middle and right light field images, the samples that are accessed during the construction of the rendered images shown on the left are colored in red, and the regions of accesses are outlined in green. In (a), an image with viewpoint near the uv camera plane is rendered. In constructing this image, the accesses are very regular and are performed on four neighboring source images in the uv plane. In (b), a closeup image with viewpoint near the st focal plane is rendered. In constructing this image, the accesses are scattered among 33 neighboring images in the uv plane. One difference between the light field visualizations on the right shown in (a) and (b) is that the red dots are scattered throughout the entire image in (a), whereas they are confined to a local region in the image in (b). This is because a limited region of (s, t) space is visible in the closeup image. Visualizations courtesy of Marc Levoy.

In the two-parallel-plane light field parameterization, images are taken from camera viewpoints that lie on a uniform 2D grid on the uv plane, with the optical axis of the camera always pointing in the normal direction to the plane. In our work, we can certainly parameterize textures (or environment maps) on the same 2D grid of camera viewpoints as used for the light field methods. However, we are not constrained to any particular parameterization. We typically parameterize the desired space of viewpoints by placing samples at regular intervals along each dimension of the space. For example, for a desired 3D space of viewpoints, we place samples on a uniform 3D grid. We can also constrain the space of viewpoints to an arbitrary 2D surface, such as a hemisphere, or an arbitrary 1D line, such as an arc. We reconstruct a novel view using either the nearest-neighbor viewpoint sample or higher-order interpolation on the neighborhood of viewpoint samples.

Apart from the fact that we store textures instead of images, which has advantages described in Chapter 4, there are two important differences between light field methods and the parameterized texture methods described in this thesis. First, to reconstruct a particular image, we guarantee that all samples accessed come from a single texture image, or a few neighboring ones when performing interpolation. This property allows us to obtain better memory coherence, and allows for more efficient compression. Better compression results from the fact that we can exploit the maximum amount of coherence available within each texture image since texture maps are atomically decoded and loaded into the hardware memory. In contrast, because the sample accesses for light field methods can be scattered requiring support for fast random lookups into any image, it becomes harder to exploit coherence within each individual image in the uv plane of camera viewpoints while also providing support for fast decompression. Note that in the case that computation is free, we can compress the light field exploiting maximum coherence, since decoding cost would not be an issue, and there would be no difference in compression efficiency with our parameterized texture methods.

We now describe the second important difference between light field methods and our parameterized texture methods. One property of light field and lumigraph methods is that only a 2D array of images is needed to reconstruct an image anywhere within a 3D space free of occluders (free space). In contrast, our approach requires a 3D array of textures to reconstruct an image anywhere within a 3D space. Thus, our improved access pattern comes at the cost of an additional dimension when the desired viewing space is a 3D space. However, our parameterization of environment maps rather than texture maps, as in Chapter 5, mitigates the need for an additional dimension since it allows for plausible reconstruction away from sampled viewpoints, such as viewpoints closer and farther from a plane of sample viewpoints.

Wong et al. [Won97] and Nishino et al. [Nis99] have extended the 4D light field to a 5D field that permits changes to the lighting environment. The challenge of such methods is efficient storage of the high-dimensional image fields.

Shum and He [Shu99] simplified the 4D representation further to a 3D field by constraining the camera motion to planar concentric circles, called concentric mosaics. This reduction in dimensionality is the main advantage of concentric mosaics as it significantly reduces the amount of data storage needed.

The *Layered depth image* (LDI) [Sha98, Cha99] is another representation of the radiance field better able to handle disocclusions without unduly increasing the number of viewpoint samples. A layered depth image stores depth in addition to color at each pixel location. It is constructed by warping multiple images with per-pixel depth into a common camera view. The representation is layered in that multiple pixels with distinct depth values can occupy a single pixel location. A desired view is constructed by warping the pixels in the LDI with back to front splatting. Like the view interpolation method of Chen and Williams [Che93], the LDI does not handle view-dependent variations in scene appearance.

For spatially coherent scenes, Miller et al. [Mil98a], Nishino et al. [Nis99] and Wood et al. [Woo00] observed that geometry-based surface fields better capture coherence in the light field and achieve a more efficient encoding than view-based images like the LDI or lumigraph. *Surface light fields* [Mil98a, Woo00] parameterize the radiance field over surfaces rather than views. Specifically, the radiance field is represented as a dense sampling over surface points of low-resolution lumispheres. Rendering involves finding the point of intersection of a viewing ray with the surface geometry, reflecting the incoming viewing ray about the normal at that point, and indexing into the appropriate lumisphere with the reflected direction. Surface light fields are especially well-suited when surfaces are mostly diffuse due to practical resolution constraints on lumispheres.

Like view-based light field methods, surface light fields may visit an irregular scattering of samples over the entire 4D light field to reconstruct a particular view, and lack hardware acceleration. In fact, samples are more likely to be scattered for surface light field methods because the accesses take the normal at each surface point into consideration when indexing into the 4D representation. Varying normals on bumpy surfaces, for example, can lead to scattering of samples. This scattering of samples can occur even if the camera is far from the object. Both kinds of light field methods also require very high sampling densities to reconstruct specular objects. In comparison, we achieve mirror-like reflections with parameterized environment maps, as described in Chapter 5.

Heidrich et al. [Hei99a] decouple geometry from illumination by using a light field to map incoming view rays into outgoing reflected or refracted rays. These outgoing rays then index either a static environment map, which ignores local effects further from the reflector, or another light field representing the environment, which is more accurate but also more costly. The result allows independent change to the reflecting object geometry and the environmental radiance, but suffers from the limitations of other IBR methods mentioned above.

Cabral et al. [Cab99] also decouple the reflecting object from the illumination. They store a collection of view-dependent environment maps where each environment map pre-integrates a specific BRDF with a lighting environment. The lighting environments for these environment maps are generated using standard techniques, such as taking photographs of a physical sphere in a desired environment or rendering the six faces of a cube from the reflecting object center using a ray-tracer. As a result these environment maps exhibit the same problems as traditional environment maps in ignoring local reflections and refractions. In contrast, with parameterized environment maps, we are able to capture local effects, like self-reflections and parallax in the reflected imagery.

Lischinski and Rappoport [Lis98] ray trace through a collection of view-dependent LDIs for glossy objects with fuzzy reflections, and three view-independent LDIs representing the diffuse environment. Bastos et al. [Bas99] reproject LDIs into a reflected view for rendering primarily planar glossy surfaces in architectural walkthroughs. Agrawala et al. [Agr00] use an LDI storing depth and attenuation to simulate soft shadows. Our approach succeeds with simpler and hardware-supported texture and environment maps rather than LDIs, resorting to ray tracing only for the local "lens" geometry where it is most necessary, as described in Chapter 6 on hybrid rendering.

Another IBR hybrid uses *view-dependent textures* (VDT) [Deb96, Deb98a, Coh99] in which geometric objects are texture-mapped using a projective mapping from view-based images. VDT methods depend on viewpoint movement for proper antialiasing – novel views are generated by reconstructing using nearby views that see each surface sufficiently "head-on". Such reconstruction is incorrect for highly specular surfaces. We instead infer texture maps that produce antialiased reconstructions independently at each parameter location, even for spaces with no viewpoint dimensions. This is accomplished by generating per-object segmented images in the ray tracer and inferring textures that match each segmented layer. In addition to our generalized parameterization, a major difference in our approach is that we use "intrinsic" texture parameterizations (i.e., viewpoint-independent (u,v) coordinates per vertex on each mesh) rather than view-based ones. We can then capture the view-independent lighting in a single texture map rather than a collection of views to obtain better compression.

One characteristic of VDT methods is that in a particular novel view, different visible surfaces can have different "best" source images. This happens, for example, when an area of a novel image is not entirely visible in any of the source images due to occlusions. Debevec et al. [Deb98a] store a view map for each polygon which identifies the most appropriate source image for a given view. The view map information is recorded over a regularly sampled space of viewing directions to support all possible views. Even with many source images, some portions of a novel view may not be visible in any of the source images. Hole filling is handled in a post-processing pass which assigns colors to polygon vertices that are closest to ones that are visible in the source images, and Gouraud shading is used to fill the interior. In contrast, disocclusions are handled in our work without encoding which polygons are visible in which views or gathering polygons corresponding to different views in separate passes. We infer a texture map per object at each pre-rendered image sample and use a pyramidal regularization term in our texture inference (Chapter 3) that provides smooth "hole-filling" for occluded regions without a specialized post-processing pass. In addition, our texture inference approach supports solving simultaneously across multiple viewpoints to eliminate disocclusion holes (Chapter 6). At runtime, we linearly interpolate between the texture (or environment) maps that correspond to the nearest views, and the same set of texture maps are accessed across the entire surface of each object as mentioned previously.

Buehler et al. [Bue01] describe an unstructured lumigraph rendering (ULR) approach that provides a generalized framework for view-based IBR approaches, having view-dependent textures and the lumigraph/light field methods as extremes. ULR takes as input an unstructured collection of input images and any geometric information known about the scene. Rendering involves computing a "camera blending field" over the whole image which specifies the weights given to each source image at each destination image pixel. These weights, computed at a sparse set of vertices in the image plane, consider factors such as the angular difference between the desired ray and those available in the source images, estimates of undersampling and field-of-view. Typically, only a few input images have non-zero weights in any given region of the image, and projective texture-mapping graphics hardware can be used for efficient rendering. However, determining the set of images that are relevant at each blending vertex requires computing the blending weights with respect to all source images, a computationally intensive task that tends to be the performance bottleneck of the approach. Moreover, although only a few input images contribute to a local region, many source images may be accessed in the process of rendering an entire image (in the worst case the entire set of source images), which reduces memory coherence and complicates decoding from compressed representations.

In summary, image-based methods are particularly useful in applications where the geometry of the environment is not known, such as real-world imagery captured using a real camera. Our work instead focuses on synthetic imagery where we have complete knowledge of the geometry and material properties of the objects being rendered. We seek to exploit this information to more efficiently encode the image space. Another difference is that we generalize to images spaces with arbitrary parameters, not just viewpoint.

2.2 Hardware Shading Models

Another approach to interactive photorealism seeks to improve hardware shading models rather than fully tabulating incident or emitted radiance.

Diefenbach [Dief96] used shadow volumes and recursive hardware rendering to compute approximations to global rendering. Accurate reflections on planar surfaces are achieved by mirroring the viewpoint about the reflection plane. Ofek and Rappoport [Ofek98] extended this work to curved reflectors by transforming each vertex in the reflected image with respect to the reflector's geometry. This scheme handles smooth reflecting objects that are either concave or convex; objects with mixed convexity or saddle regions require careful decomposition.

Kautz and McCool [Kau99] and McCool et al. [McC01] assumed point light sources and factored the BRDF to compute texture maps that can be used for hardware rendering at per-pixel resolution. Ramamoorthi and Hanrahan [Ram01a] showed that the irradiance for diffuse objects under distant illumination is well approximated by an analytic expression with just nine coefficients, and demonstrated rendering at real-time rates with programmable vertex shading.

Miller et al. [Mil98b] described rendering optimizations using graphics hardware, such as caching data for faster evaluation of bump-mapped surfaces and lighting, and showed how animated bump maps on planar surfaces can be used to simulate reflective ripples in a water simulation. Heidrich and Seidel [Hei99b] encoded anisotropic lighting and specular reflections with Fresnel effects using hardware texturing. Heidrich et al. [Hei00] simulated self-shadowing and indirect scattering effects in height fields and bump maps using precomputed visibility information and multi-texturing in hardware.

Nimeroff et al. [Nim94] and Teo et al. [Teo97] efficiently rendered images with novel lighting conditions by summing a linear combination of pre-rendered basis images. Malzbender et al. [Mal01] fit polynomials with six coefficients to images of an object taken from a fixed viewpoint with varying illumination. The polynomials can be evaluated in real-time on graphics hardware using programmable texture operations. They effectively capture view-independent effects like self-shadowing and diffuse shading that depend on the illumination direction relative to the object, but exclude view-dependent effects such as specularity.

Udeshi and Hansen [Ude99] exploited general-purpose CPU and graphics hardware parallelism to interactively render indoor scenes with photorealistic effects such as soft shadows and indirect illumination. Lengyel et al. [Len00, Len01] showed how texturing hardware can be used for real-time rendering of furry models.

Even using many parallel graphics pipelines (8 for [Ude99]) these approaches can only handle simple scenes, and, because of limitations on the number of passes, do not capture all the effects of a full offline photorealistic rendering, including multiple bounce reflections and refractions and accurate shadows.

2.3 Texture Recovery/Model Matching

The recovery of texture maps from images is closely related to surface reflectance estimation in computer vision [Sat97, Mar98, Yu99, Ram01b]. Yu et al. [Yu99] recover diffuse albedo maps and a spatially invariant characterization of specularity in the presence of unknown, indirect lighting. We greatly simplify the problem by using known geometry and separating diffuse and specular lighting layers during the offline rendering. We focus instead on the problem of inferring textures for a particular graphics hardware that "undo" its undesirable properties, like poor-quality texture filtering. A related idea is to compute the best hardware lighting to match a gold standard [Wal97].

2.4 Compression

Various strategies for compressing the dual-plane lumigraph parameterization have been proposed. Levoy [Lev96] used vector quantization and entropy coding to get compression ratios of up to 118:1 while Lalonde and Fournier [Lal99] used a wavelet basis with compression ratios of 20:1. Magnor [Mag00] studied compression algorithms with various levels of reconstructed geometric information, and achieved 200:1 compression at good quality using a hierarchical disparity-compensated coder. Similar results were also reported by Tong and Gray [Ton00, Ton01] with an emphasis on fast decoding.

Miller et al. [Mil98a] compressed the 4D surface light field using a block-based DCT encoder with compression ratios of 20:1. Nishino et al. [Nis99] used an eigenbasis (K-L transform) to encode surface textures achieving compression ratios of 20:1 with eigenbases having 8-18 texture vectors. Such a representation requires an excessive number of "eigentextures" to faithfully encode highly specular objects. This prohibits real-time decoding, which involves computing a linear combination of the eigentextures. Wood et al. [Woo00] compressed the surface light field using techniques similar to vector quantization and principal component analysis and achieved 70:1 compression.

Concentric mosaics have been compressed at compression ratios up to 120:1 [Zha00, Wu00]. Other work on texture compression in computer graphics includes Beers et al. [Bee96], who used vector quantization on 2D textures for compression ratios of up to 35:1.

We use a Laplacian pyramid on blocks of the parameter space. This speeds run-time decoding (for 8×8 blocks of a 2D parameter space, only 4 images must be decompressed and added to decode a texture) and achieves good quality at compression ratios up to 800:1 in our experiments.

Another relevant area of work is animation compression. Standard video compression uses simple block-based transforms and image-based motion prediction [Leg91]. Guenter et al. [Gue93] observed that compression is greatly improved by exploiting information available in synthetic animations. In effect, the animation script provides perfect motion prediction, an idea also used in [Wal94, Agr95]. Levoy [Lev95] showed how simple graphics hardware could be used to match a synthetic image stream produced by a simultaneously-executing, high-quality server renderer by exploiting polygon rendering and transmitting a residual signal to the client. Cohen-Or et al. [Coh99] used view-dependent texture maps to progressively transmit diffusely-shaded, texture-intensive walkthroughs, finding factors of roughly 10 improvement over MPEG for scenes of simple geometric complexity. We extend this work to the matching of multidimensional animations containing non-diffuse, offline-rendered imagery by texture-mapping graphics hardware.

Chapter 3

Texture Inference by Inverse Rendering

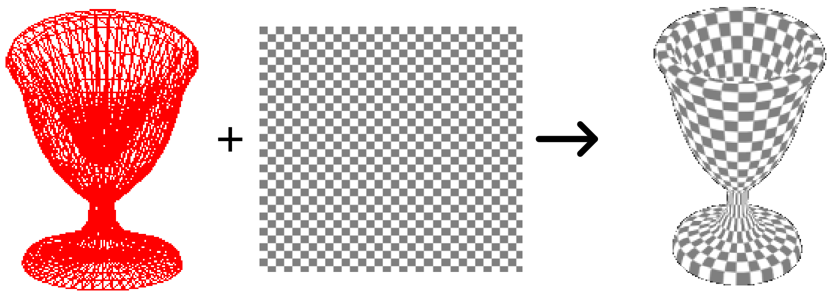
Texture Mapping, illustrated in Figure 3(a), refers to the process of mapping a 2-dimensional texture image to a 3-dimensional model of an object. In our illustration, a texture composed of a checkerboard pattern is mapped onto the geometric model of a cup. A critical task performed by graphics hardware when texture mapping is filtering the texture image to prevent aliasing on the screen.

Texture inference, illustrated in Figure 3(b), inverts the texture mapping process, in that we start with a 3-dimensional geometric model and a 2-dimensional offline rendered image that we would like to match, and solve for a texture. To *infer* a texture map means to find one which when applied to a hardware-rendered geometric object matches the offline-rendered image. The key observation to obtain an accurate match is to take the hardware filter model into account.

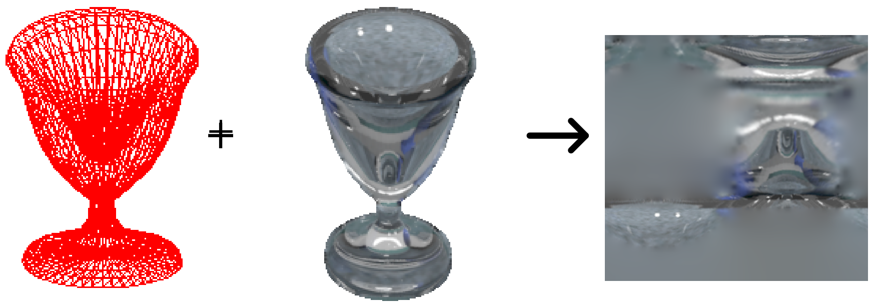
In this chapter, we consider the input images in the parameterized image space separately, and infer texture maps for individual objects in the scene from each image. We begin by segmenting the input images (Section 3.1), and choose an appropriate texture domain and resolution for each object (Section 3.2). We then briefly describe three possible approaches to compute texture maps (Section 3.3). To best match the input rendered images, we model the graphics hardware as a large sparse linear system (Section 3.4), and perform a least-squares optimization on the resulting system (Section 3.5). We conclude this chapter by comparing the accuracy of matching to the input images across texture inference algorithms and hardware filter models (Section 3.6).

3.1 Segmenting Ray-Traced Images

Each geometric object has a parameterized texture that must be inferred from the ray-traced images. These images are first segmented into per-object pieces to prevent bleeding of information from different objects across silhouettes. Bleeding decreases coherence and leads to misplaced silhouettes when the viewpoint moves away from the original samples. To perform per-object segmentation, the



(a) Texture Mapping

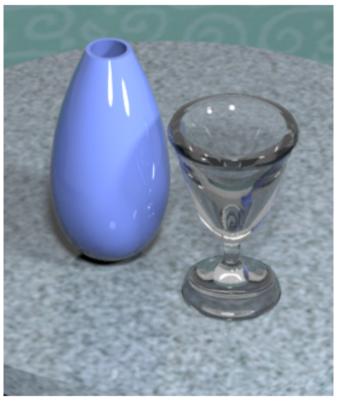


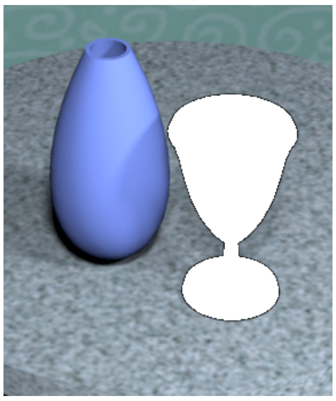
(b) Texture Inference

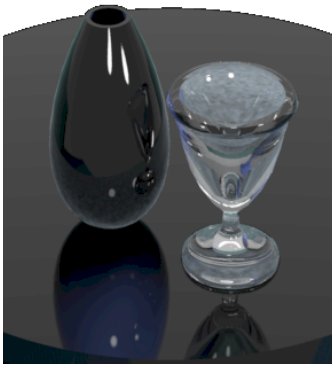
Figure 3.1: **Texture Mapping and Texture Inference.**

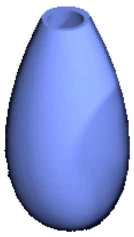
ray tracer generates a per-object mask as well as a combined image, all at supersampled resolution. For each object, we filter the portion of the combined image indicated by the mask and divide by the fractional coverage computed by applying the same filter to the object's mask. A gaussian filter kernel is used to avoid problems with negative coverages.

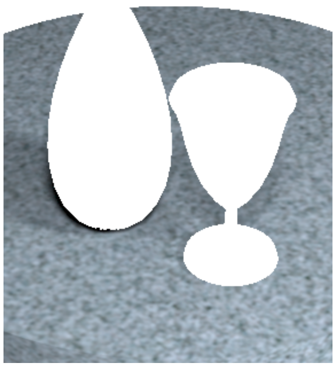
A second form of segmentation separates the view-dependent specular information from the view-independent diffuse information for the common case that the parameter space includes at least one view dimension. This reduces the dimensionality of the parameter space for the diffuse layer, improving compression. As the image is rendered, the ray-tracer places information from the first diffuse intersection in a view-independent layer and all other information in a view-dependent one. Figure 3.1 illustrates segmentation for an example ray-traced image. We use a modified version of Eon, a Monte Carlo distribution ray-tracer [Coo84a, Shi92, Shi96].


(a) Complete Image


(b) Diffuse Layer

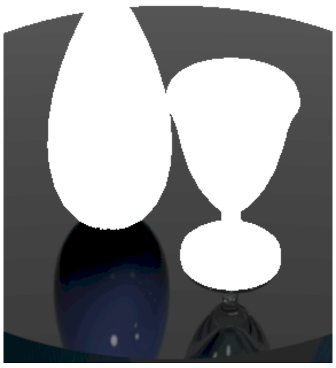

(c) Specular Layer


(d) Diffuse Vase Layer


(e) Diffuse Table Layer

(f) Diffuse Wall Layer

(g) Specular Vase Layer


(h) Specular Table Layer

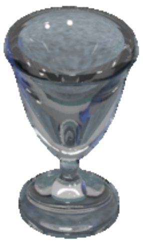

(i) Specular Cup Layer

Figure 3.2: **Segmentation of Ray-Traced Images.**

3.2 Optimizing Texture Coordinates and Resolutions

Since parts of an object may be occluded or off-screen, only part of its texture domain is accessed. In this section, we are given as input the geometric mesh for an object together with parameter-independent (u, v) texture coordinates per vertex. The overall problem that we are trying to solve is to find the minimum texture resolution needed for a particular region of the parameter space. The original texture coordinates of the geometry are used as a starting point and then optimized so as to: 1) to ensure adequate sampling of the visible texture image with as few samples as possible, 2) to allow efficient computation of texture coordinates at run-time, and 3) to minimize encoding of the optimized texture coordinates. To satisfy the last two goals, we choose and encode a parameter-dependent affine transformation on the original texture coordinates rather than re-specify them at each vertex. One affine transformation is chosen per object per block of parameter space (see Chapter 4). Just six values are required for each object's parameter space block and texture coordinates can be computed with a simple, hardware-supported transformation. The algorithm follows:

```
1  Reposition branch cut in texture dimensions that have wrapping enabled
2  Find least-squares most isometric affine transformation
3  Compute maximum singular value of Jacobian of texture to screen space
       mapping and scale transformation along direction of maximal stretch
4  Repeat 3 until maximum singular value is below a given threshold
5  Identify bounding rectangle with minimum area
6  Determine texture resolution
```

We first attempt to reposition the (u, v) texture coordinate origin in any texture dimensions that are periodic (i.e., have wrapping enabled). A separate "branch cut" is performed in each dimension in which the origin is repositioned. This adjustment realigns parts of the visible texture domain that have wrapped around to become discontiguous, for example, when the periodic seam of a cylinder becomes visible. A smaller portion of texture area can then be encoded. We consider each of the *u* and *v* dimensions independently, and compute the texture coordinate extents of visible triangle edges after clipping with the viewing frustum. If a gap in the visible extents exists, a branch cut is performed and texture wrapping disabled for that dimension. A branch cut is computed as follows. Assuming a branch cut is performed in texture dimension w (referring to either u or v), the new position of the origin in the w dimension is w_{cut}, and the w-coordinate at a particular vertex i is w_i, let $w'_i = w_i - w_{cut}$. The new w-coordinate for vertex i is computed as $w'_i - \lfloor w'_i \rfloor$.

We then find the linear transformation, $R(u, v)$, minimizing the following objective function, inspired by [Mai93]

$$
\begin{aligned}
R(u,v) &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \\
f(x) &= \sum_{\text{edges } i} W_i \left(\frac{s_i - \|R(u_{i_0}, v_{i_0}) - R(u_{i_1}, v_{i_1})\|}{\min\left(s_i, \|R(u_{i_0}, v_{i_0}) - R(u_{i_1}, v_{i_1})\|\right)} \right)^2
\end{aligned} \tag{3.1}
$$

where s_i represents the length on the screen of a particular triangle edge, i_0 and i_1 represent the edge vertices, and W_i is a weighting term which sums screen areas of triangles on each side of the edge. The sum in f is taken over *visible* triangle edges at each point in the parameter block, determined by rasterizing triangle identifiers into a zbuffer to select visible triangles. Visible triangle edges are also clipped to the view frustum.

Note that when a branch cut is not possible in the previous step over a “wrapped” or periodic dimension, we reduce the affine transformation to a scale transformation by fixing the values of b and c to zero. This ensures that the texture’s periodic boundary conditions are not disturbed. Note also that the translational components of the affine transformation cancel from the objective function.

This minimization choses a mapping that is as close to an isometry as possible by minimizing length difference between triangle edges in texture space and projected to the image. We divide by the minimum edge length so as to equally penalize edges that are an equal factor longer and shorter. $\nabla f(x)$ is calculated analytically for use in conjugate gradient minimization. Note that a rotational degree of freedom remains in this optimization which is fixed in step 5.

In the third step, we ensure that the object’s texture map contains enough samples by scaling the R found previously. We check the greatest local stretch (singular value) across all screen pixels in which the object is visible, using the Jacobian of the mapping from texture to screen space. Since the Jacobian for the perspective mapping is spatially varying even within a single polygon, this computation is performed separately at each screen pixel. If the maximum singular value exceeds a user-specified threshold (such as 1.25), we scale R by the maximum singular value in the corresponding direction of maximal stretch, and iterate until the maximum singular value is reduced below the threshold. This essentially adds more samples to counteract the worst-case stretching of the projected texture.

The next step identifies the minimum-area bounding rectangle on the affinely transformed texture coordinates by searching over a set of discrete directions. The size of the bounding rectangle also determines the optimal texture resolution.

We can now pick the actual texture resolution. Since present-day graphics hardware typically constrains the texture resolution to powers of two, we conservatively pick the nearest power of two that is larger than the optimal texture resolution, but only use the part of the texture that is actually needed. We keep a record of the optimal texture resolution so that only the portion of the texture which is used is compressed.

Because the optimal texture resolution is arbitrary (i.e. not necessarily a power of 2), and regions of the texture that are outside the optimal texture resolution are undefined, we must take care in computing Mip Maps. Specifically, we must support computing Mip Maps for arbitrarily sized textures both in the texture inference solver and in the runtime system to avoid filtering undefined values into the Mip Map. Assuming a bilinear 2×2 filter is used to compute the filtered Mip Map levels, this can be achieved by extending the texture by one texel, essentially replicating the boundary texels, for each level in the Mip Map where the texture resolution is odd prior to filtering.

Finally, since texture resolution substantially impacts performance due to texture decompression and transfer between system and video memory, our compiler accepts user-specified resolution reduction factors that scale the optimal texture resolution on a per-object basis.

3.3 Approaches for Computing Texture Maps

Three different approaches for computing texture maps are discussed below. We present quantitative comparisons in the quality of rendered images between these approaches in Section 3.6.

- **Forward Mapping Method for Texture Inference**
  A simple texture inference algorithm maps each texel to the image and then filters the neighboring region to reconstruct the texel's value [Mar98]. This method is illustrated in Figure 3.3.

  One problem with this approach is reconstruction of texels near arbitrarily-shaped object boundaries and occluded regions (Figure 3.1d-i). Such occluded regions produce undefined texture samples which complicates building of MIPMAPs. Finally, the simple algorithm does not take into account how texture filtering is performed on the target graphics hardware.

- **Least-Squares Method for Texture Inference**
  A more principled approach, discussed in more detail in the next two sections, is based on the observation that a texture pixel contributes to zero or more display pixels. Neglecting quantization effects, a texture pixel that is twice as bright contributes twice as much. Thus, we can model the hardware texture mapping operation in the form of a linear system, $Ax = b$, where matrix A represents the hardware filter coefficients mapping texels to display pixels, vector x represents the texture to be solved for, and vector b represents the ray-traced image to be matched. We determine elements of the matrix A by performing test renderings on the

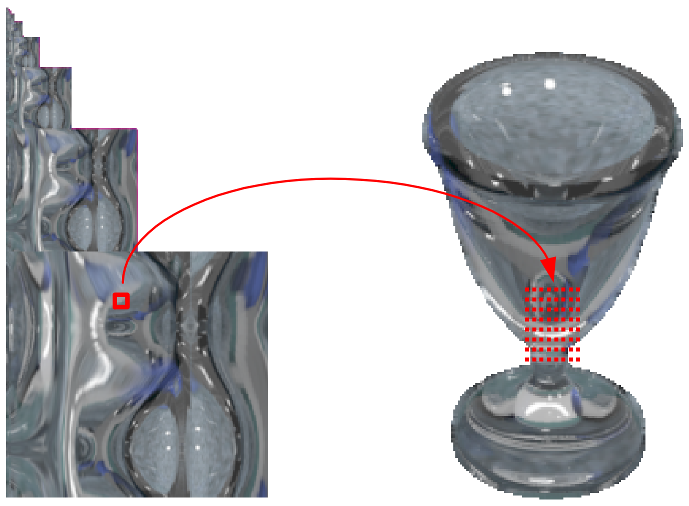


Figure 3.3: **Forward Mapping.** Each texture sample is mapped to the screen, and its value computed by performing high-quality filtering on the neighboring region in the ray-traced image.

hardware that isolate the contribution of each texel. Considering that the matrix A is sparse, we can solve for x using conjugate gradient method.

This least-squares approach considers the texture mapping filter model used by hardware to best match the hardware rendering to the input images. Moreover, we introduce a pyramidal regularization term in the optimization that ensures the entire MIPMAP texture is defined, with occluded regions filled smoothly.

- **Ray-Tracing Directly into Texture Maps**
  A third approach sidesteps the problem of inferring texture maps from images by ray-tracing directly into texture maps. Rather than casting rays from the eye-point through a uniformly sampled plane in 3D, we instead map texel centers to the image and cast rays towards these locations, writing the eventual values for the rays directly into the texel locations in the texture maps. To avoid aliasing artifacts in the texture, we supersample the texture image, shooting independent rays for each sub-texel, and use a high-quality filter to down-sample to the final texture resolution.

  One problem with this approach is that texture samples can map onto back-facing regions of the object. As with the forward mapping method, such undefined texture samples can complicate building of MIPMAPs. Another drawback of this approach is that ray-tracing into textures is generally more costly than ray-tracing into an image. This is because when ray-tracing into a texture, the sampling density in the image plane is a function of the mapping from the texture to the screen. To ensure adequate sampling in regions of the object that are seen head-on, the texture resolution must be sufficiently high. This results in an overly dense sampling in regions of the object that are seen at an angle, such as at the silhouettes. In contrast, when rendering into an image, the sampling density is determined by the screen resolution and the samples are uniformly distributed. Finally, as with the forward mapping method, this algorithm does not take into account how texture filtering is performed on the target graphics hardware.

3.4 Modeling Hardware Rendering as a Linear System

As mentioned earlier, a more principled approach is to model the hardware texture mapping operation in the form of a linear system:

$$\overbrace{\begin{bmatrix} s_{0,0} \text{ filter coefficients} \\ s_{0,1} \text{ filter coefficients} \\ s_{0,2} \text{ filter coefficients} \\ \vdots \\ s_{m-1,n-1} \text{ filter coefficients}\end{bmatrix}}^{A} \overbrace{\begin{bmatrix} \left.\begin{matrix} x^0_{0,0} \\ \vdots \\ x^0_{u-1,v-1}\end{matrix}\right\} \begin{matrix}\text{level}\\0\end{matrix} \\ \left.\begin{matrix} x^1_{0,0}\\ \vdots \\ x^1_{\frac{u}{2}-1,\frac{v}{2}-1}\end{matrix}\right\} \begin{matrix}\text{level}\\1\end{matrix} \\ \vdots \\ \left.\begin{matrix} x^{l-1}_{0,0}\\ \vdots\\ x^{l-1}_{\frac{u}{2^{l-1}}-1,\frac{v}{2^{l-1}}-1}\end{matrix}\right\} \begin{matrix}\text{level}\\l-1\end{matrix}\end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} s_{0,0}\\ s_{0,1}\\ s_{0,2}\\ \vdots\\ s_{m-1,n-1}\end{bmatrix}}^{b} \tag{3.2}$$

where vector b contains the ray-traced image to be matched, matrix A contains the filter coefficients applied to individual texels by the hardware, and vector x represents the texels from all $l-1$ levels of the MIPMAP to be inferred. Superscripts in x entries represent MIPMAP level and subscripts represent spatial location. This model ignores hardware nonlinearities in the form of rounding and quantization. All three color components of the texture share the same matrix A.

Each row in matrix A corresponds to a particular screen pixel, while each column corresponds to a particular texel in the texture's MIPMAP pyramid. The entries in a given row of A represent the hardware filter coefficients that blend texels to produce the color at a given screen pixel. Hardware

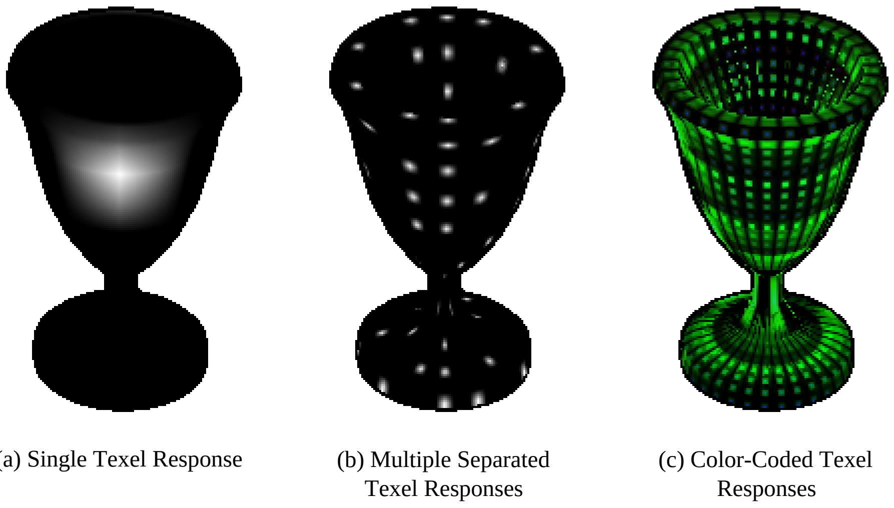

(a) Single Texel Response
(b) Multiple Separated Texel Responses
(c) Color-Coded Texel Responses

Figure 3.4: **Obtaining Matrix A.** (a) Screen image with single texel in an 8x8 texture is set to full intensity value (b) Screen image when multiple texels in a 64x64 texture image are set to full intensity values, such that alternate 8x8 blocks do not overlap. (c) Screen image with 256x256 texture where two of the color components are used for encoding texel identifiers.

filtering requires only a small number of texel accesses per screen pixel, so the matrix A is very sparse. We use hardware z-buffering to determine object visibility on the screen, and need only consider rows (screen pixels) where the object is visible. Other rows are logically filled with zeroes but are actually deleted from the matrix, by using a table of visible pixel locations. Filter coefficients should sum to one in any row.

In practice, row sums of inferred coefficients are often less than one due to truncation errors. A simple correction is to add an appropriate constant to all nonzero entries in the row. A more accurate method recognizes that each coefficient represents the slope of a straight line in a plot of screen pixel versus texel intensity. We can therefore test a variety of values and return the least squares line. We use the first method because it is much faster.

3.4.1 Obtaining Matrix A

A simple but impractical algorithm for obtaining A examines the screen output from a series of renderings, each setting only a single texel of interest to a nonzero value, as follows

```
Initialize z-buffer with visibility information by rendering entire scene
For each texel in MIPMAP pyramid,
    Clear texture, and set individual texel to maximum intensity
    Clear framebuffer, and render all triangles that compose object
    For each non-zero pixel in framebuffer,
        Divide screen pixel value by maximum framebuffer intensity
        Place fractional value in $A$[screen pixel row][texel column]
```

Accuracy of inferred filter coefficients is limited by the color component resolution of the framebuffer, typically 8 bits.

3.4.2 Parallel Matrix Inference with Non-overlapping Bounding Boxes

To accelerate the simple algorithm, we observe that multiple columns in the matrix A can be filled in parallel as long as texel projections do not overlap on the screen and we can determine which pixels derive from which texels. An algorithm that subdivides texture space into blocks and checks that alternate texture block projections do not overlap can be devised based on this observation as follows

```
Initialize z-buffer with visibility information by rendering entire scene

Part 1: Find screen space bounding boxes for visible blocks
Pick texture block size that minimizes overall cost
Perform quadtree recursion down to chosen block size
For each visible texture block,
    Clear texture, and set all texels in block to maximum intensity
    Clear framebuffer, and render all triangles that compose object
    Compute bounding box on screen of nonzero pixels

Part 2: Infer Matrix Coefficients in Parallel
Repeat until all texture blocks have been processed,
    Pick blocks with non-overlapping bounding boxes on screen
    For each texel position in block,
        Clear texture, and set exactly one texel in each block to maximum intensity
        Clear framebuffer, and render all triangles that compose object
        For each non-zero pixel in framebuffer,
            Divide screen pixel value by maximum framebuffer intensity
            Place fractional value in $A$[screen pixel row][texel column]
                where texel column is determined from bounding box data
```

We pick the block size to minimize the overall cost of finding the screen space bounding boxes and inferring the individual columns of matrix coefficients. This block size is computed as

$$bs = \text{closest power of 2 of } \sqrt[4]{\frac{ts^2}{4}} \tag{3.3}$$

where bs represents block size, and ts represents texture size. One optimization in the first part of the algorithm is to perform quadtree recursion on the entire texture to quickly eliminate regions of the texture that are not used by any visible pixels.

3.4.3 Parallel Matrix Inference with Texel Identifiers

A better algorithm recognizes that since just a single color component is required to infer the matrix coefficients, the other color components (typically 16 or 24 bits) can be used to store a unique texel identifier that indicates the destination column for storing the filtering coefficient. The outline for this algorithm follows

```
Initialize z-buffer with visibility information by rendering entire scene
Loop over three MIPMAP levels,
    Loop over 6x6 renderings (6x8, 8x6, or 8x8 with periodic dimensions),
        Initialize all texels with appropriate texel identifiers
        Clear texture, and set texels considered in parallel to maximum intensity
        Clear framebuffer, and render all triangles that compose object
        For each nonzero pixel in framebuffer,
            Divide screen pixel value by maximum framebuffer intensity
            Place fractional value in $A$[screen pixel row][texel column]
                where texel column is determined using texel identifier value
                found in screen pixel
```

The details for initializing the texels with the appropriate texel identifiers are discussed below. With this algorithm, the matrix A can be inferred in 108 renderings for trilinear MIPMAP filtering, independent of texture resolution. The essential constraint on the algorithm is that no two texels solved in parallel can contribute to the same screen pixel. This constraint happens because otherwise it would not be possible to discern the individual contribution of each texel to the screen output. We enforce this constraint by "adequate" skipping of texels between those that are solved in parallel in each rendering, and combine results across multiple renderings to infer the complete matrix A. The number of 108 renderings is obtained by solving in parallel every sixth sample in both dimensions of the same MIPMAP level, and every third MIPMAP level, thus ensuring that possible filtering neighborhoods of samples solved in parallel do not interfere. For hardware with the power of two constraint on texture resolution, there is an additional technical difficulty when the texture map has one or two periodic (wrapping) dimensions. In that case, since 6 does not evenly divide any power

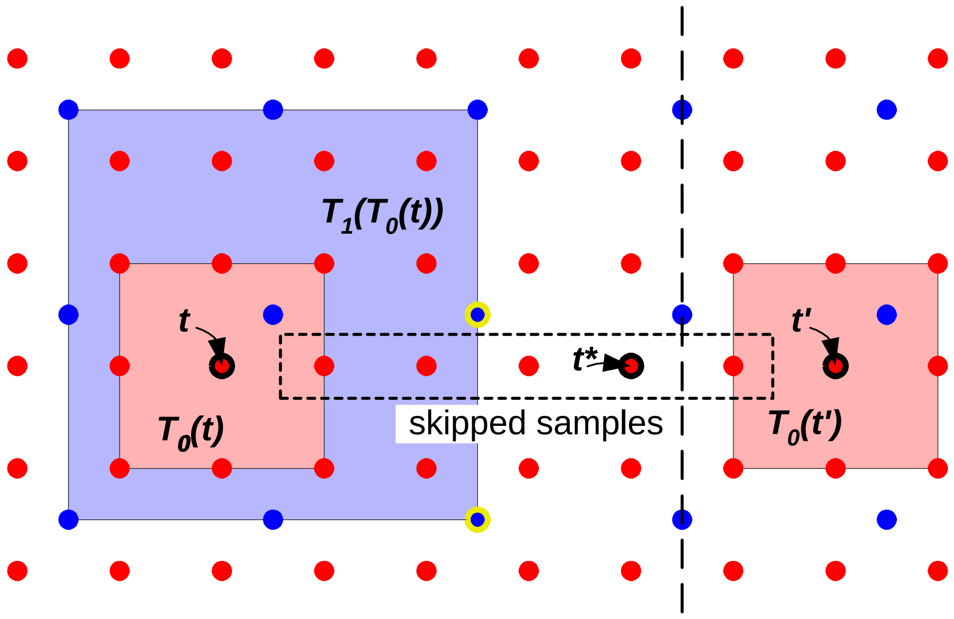


Figure 3.5: **Trlinear Filtering Neighborhood for Parallel Matrix Inference.** Red dots represent texel samples; blue dots are samples in the next higher level of the MIPMAP. See text for explanation.

of 2, the last group of samples may wrap around to interfere with the first group. One solution simply solves in parallel only every eighth sample.

For trilinear MIPMAP filtering, a given screen pixel accesses four texels in one MIPMAP level, as well as four texels either one level above or below having the same texture coordinates. By leaving sufficient spacing between texels computed in parallel, A can be inferred in a fixed number of renderings, P, where P=6x6x3=108. This assumes that the extra color components contain at least $log_2(n/P)$ bits where n is the number of texels.

In Figure 3.4.3, red dots represent texel samples; blue dots are samples in the next higher level of the MIPMAP. To infer the filter coefficients at a sample t, we must ensure that all samples that could possibly be filtered with it to produce a screen pixel output have identical texel identifiers. The region $T_0(t)$ represents the region of texture space in the same MIPMAP level that could possibly access sample t with bilinear filtering, called its *level 0 neighborhood.* This region can possibly access samples from the next higher level of the MIPMAP shown in blue and labeled $T_1(T_0(t))$, the level 1 neighborhood of t's level 0 neighborhood. We must not solve in parallel a texel that shares any of these samples in its filtering neighborhood so only texels whose level 0 neighborhood are completely to the right of the dashed line are candidates. For example, the sample labeled t^* can not be solved in parallel with t since t^*'s level 1 neighborhood shares two samples with t, shown outlined in yellow. Even the sample to its right must be skipped since its level 0 neighborhood still includes shared samples at the next higher MIPMAP level. Sample t' is the closest to t that can be solved in parallel. Thus in each dimension, at least 5 samples must be skipped between texels that are solved in parallel.

To avoid corrupting the identifier for each texel that is being solved, we must store the same texel identifier in the possible filtering neighborhood of a texel. The filtering neighborhood of a texel consists of the level 0 and level 1 neighborhoods that are labeled as $T_0(t)$ and $T_1(T_0(t))$, respectively, in Figure 3.4.3, as well as the texels that can possibly be accessed in the same filter operation at the next lower level of the MIPMAP (not shown in the figure). The filtering neighborhoods for two texels solved in parallel are guaranteed not to overlap because of the adequate skipping of texels discussed above. Thus, we can easily determine what identifier to assign to each texel in the entire texture map.

3.4.4 Inference with Antialiasing

To antialias images, we can perform supersampling and filtering in the graphics hardware. Unfortunately, this decreases the precision with which we can infer the matrix coefficients, since the final result is still an 8-bit quantity in the framebuffer. Higher precision is obtained by inferring based on the supersampled resolution (i.e. without antialiasing), and filtering the matrix A using a higher-precision software model of the hardwares antialiasing filter. Sub-pixels (rows in the supersampled matrix) that are not covered by the object should not contribute to the solution. As in the segmentation technique of Section 3.1, we filter the matrix A and then divide by the fractional coverage at each pixel as determined by the hardware rendering. Small errors arise because of minor differences in pixel coverage between the ray-traced and hardware-generated images.

3.5 Least-Squares Solution

Removing irrelevant image pixels from Equation (3.2), A becomes an $n_s \times n_t$ matrix, where n_s is the number of screen pixels in which the object is visible, and n_t is the number of texels in the object's texture MIPMAP pyramid. Once we have obtained the matrix A, we solve for the texture represented by the vector x by minimizing a function $f(x)$ defined via

$$\begin{aligned} f(x) &= \|Ax - b\|^2 \\ \nabla f(x) &= 2A^T(Ax - b) \end{aligned} \tag{3.4}$$

subject to the constraint $0 \leq x_{i,j}^{k} \leq 1$. Given the gradient, $\nabla f(x)$, the conjugate gradient method can be used to minimize $f(x)$. The main computation of the solution's inner loop multiplies A with a vector x representing the current solution estimate and, for the gradient, A^T with $Ax - b$. Since A is a sparse matrix with each row containing a small number of nonzero elements (exactly 8 with trilinear filtering), the cost of multiplying A or A^T with a vector is proportional to n_s. Another way to express $f(x)$ and $\nabla f(x)$ is:

$$\begin{aligned} f(x) &= xA^T Ax - 2x \cdot A^T b + b \cdot b \\ \nabla f(x) &= 2A^T Ax - 2A^T b \end{aligned} \tag{3.5}$$

In this formulation, the inner loop's main computation multiplies $A^T A$, an $n_t \times n_t$ matrix, with a vector. Since $A^T A$ is also sparse, though less so than A, the cost of multiplying $A^T A$ with a vector is proportional to n_t. We use the following heuristic to decide which set of equations to use:

```
if ($2n_s \geq Kn_t$)
    Use $A^T A$ method: Equation (3.5)
else
    Use $A$ method: Equation (3.4)
```

where K is a measure of relative sparsity of $A^T A$ compared to A. We use $K = 4$. The factor 2 in the test arises because Equation (3.4) requires two matrix-vector multiplies while Equation (3.5) only requires one.

The solver can be sped up by using an initial guess vector x that interpolates the solution obtained at lower resolution. The problem size can then be gradually scaled up until it reaches the desired texture resolution [Lue94]. This multiresolution solver idea can also be extended to the other dimensions of the parameter space. Alternatively, once a solution is found at one point in the parameter space, it can be used as an initial guess for neighboring points, which are immediately solved at the desired texture resolution. We find the second method to be more efficient.

3.5.1 Solving with Hx

Rather than modeling the hardware rendering with the matrix A, one can consider substituting the term Ax in $f(x)$ and $\nabla f(x)$ with an actual hardware rendering with texture x, denoted as Hx. In this formulation, we minimize a function $f(x)$ defined via

$$\begin{aligned} f(x) &= \|Hx - b\|^2 \\ \nabla f(x) &= 2A^T(Hx - b) \end{aligned} \tag{3.6}$$

Since graphics hardware can only handle integer texel values, the vector x must be quantized prior to hardware rendering. The advantage of this scheme is that it can lead to a more accurate solution, x, by reducing the dependence on the accuracy of the matrix A. Note that we still need to infer the matrix A to compute the gradient, $\nabla f(x)$, which is used by the conjugate gradient method; performing the A^T multiplication efficiently in hardware would require support for image convolution, a feature that is not presently available. The main disadvantage of this scheme is that it requires performing hardware rendering in the inner loop of the optimization procedure, and this can cause the solver to be slower by an order of magnitude compared with solving with Ax.

3.5.2 Solving for View-Independent Textures

Segmenting the ray-traced images into view-dependent and view-independent layers allows us to collapse the view-independent textures across multiple viewpoints. To compute a single diffuse texture, we solve the following problem:

$$\overbrace{\begin{bmatrix} A_{v_0} \\ A_{v_1} \\ A_{v_2} \\ \vdots \\ A_{v_{n-1}} \end{bmatrix}}^{A'} \begin{bmatrix} \\ \\ x \\ \\ \\ \end{bmatrix} = \overbrace{\begin{bmatrix} b_{v_0} \\ b_{v_1} \\ b_{v_2} \\ \vdots \\ b_{v_{n-1}} \end{bmatrix}}^{b'} \tag{3.7}$$

where matrix A' coalesces the A matrices for the individual viewpoints v_0 through v_{n-1}, vector b' coalesces the ray-traced images at the corresponding viewpoints, and vector x represents the diffuse texture to be solved. Since the number of rows in A' tends to be much larger than the number of columns, we use the A^TA method described earlier in Equation 3.5. In addition to speeding up the solver, this method also reduces memory requirements.

3.5.3 Regularization

Samples in the texture solution should lie in the interval [0,1]. To ensure this we add a regularizing term to the objective function $f(x)$, a common technique for inverse problems in computer vision

[Ter86, Lue94, Eng96]. The term, called the *range regularization*, is defined as follows:

$$\begin{aligned} g(x_{ij}^k) &= \frac{1}{(x_{ij}^k+\delta)(1+\delta-x_{ij}^k)} \\ f_{reg-01}(x) &= f(x)+\varepsilon_b \overbrace{\left(\frac{n_s}{n_t}\right)\left(\frac{1}{g(0)-g(1/2)}\right)}^{\text{calibration constant}} \sum_{ijk} g(x_{ij}^k) \end{aligned} \tag{3.8}$$

where $\delta = 1/512$. The function g approaches infinity at $-\delta$ and $1+\delta$ and thus penalizes texels outside the range. The regularizing term consists of three parts: a summation over all texels in x of the function g, a calibration constant giving the regularizing term roughly equal magnitude with f, and a user-defined constant, ε_b, that adjusts the importance of constraint satisfaction. We compute ∇f_{reg-01} analytically for the conjugate gradient method.

One of the consequences of setting up the texture inference problem in the form of Equation (3.2) is that only texels actually used by the graphics hardware are solved, leaving the remaining texels undefined. To support movement away from the original viewpoint samples and to make the texture easier to compress, all texels should be defined. This can be achieved by adding a second term, called the *pyramidal regularization*, of the form:

$$f_{reg-pyramid}(x) = f_{reg-01}(x)+\varepsilon_f\left(\frac{n_s}{n_t}\right)\Gamma(x) \tag{3.9}$$

where $\Gamma(x)$ takes the difference between texels at each level of the MIPMAP with an interpolated version of the next coarser level as illustrated in Figure 3.5.3. The factor n_s/n_t gives the regularization term magnitude roughly equal with f. The objective function f sums errors in screen space, while the two regularization terms sum errors in texture space, hence a scale of the regularization terms by n_s/n_t. Again, we compute $\nabla f_{reg-pyramid}$ analytically. This regularizing term essentially imposes a filter constraint between levels of the MIPMAP, with user-defined strength ε_f. We currently define Γ using simple bilinear interpolation. We find that the first regularizing term is not needed when this MIPMAP constraint is used.

3.6 Texture Inference Results

Figures 3.7-3.11 show results of texture inference on a glass cup object. In Figure 3.7, we show the texture maps that have been inferred with our least-squares texture inference approach as well as with the forward mapping and ray-tracing approaches. The top three texture maps are inferred using the least-squares inference method with three different texture filtering modes: bilinear, trilinear

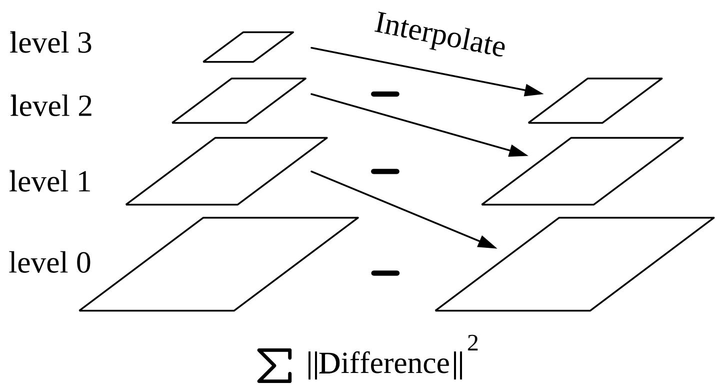


Figure 3.6: **Pyramidal regularization.** Computed by taking the sum of squared differences between texels at each level of the MIPMAP with the interpolated image of the next higher level.

and anisotropic. These three textures were solved with $\varepsilon_b = 1/32$ and did not use pyramidal regularization, resulting in many undefined texels, colored in pink. We also solved for a texture, shown in the far left of the bottom row, using a pyramidal regularization constraint with $\varepsilon_f = 0.1$ rather than a range regularization term.

Figure 3.8 compares the original image to be matched, in the upper left, with hardware-rendered images using inferred texture maps. Figure 3.9 presents close-up results for the stem region of the cup. All hardware-rendered images were generated with the Nvidia GeForce3 chip, which supports an anisotropy factor up to 4.

In general, bilinear filtering provides the sharpest and most accurate result because it uses only the finest level MIPMAP and thus has the highest frequency domain with which to match the original. Trilinear MIPMAP filtering produces a somewhat worse result, and anisotropic filtering with anisotropic factors up to 2 and up to 4 are in between. It can be seen in Figure 3.7 that more texture area is filled from the finest pyramid level for anisotropic filtering compared to trilinear, especially near the cup's stem, while bilinear filtering altogether ignores the higher MIPMAP levels. Note though that bilinear filtering produces this highly accurate result *only at the exact parameter values (e.g., viewpoint locations) and image resolutions where the texture was inferred.* The other schemes are superior if viewpoint or image resolution are changed from those samples.

The last two columns in the middle row of Figure 3.8 compare the effect of solving with pyramidal regularization. Both images were generated with up to a factor of 4 anisotropic filtering. It can be seen that the images are almost identical with $\varepsilon_f = 0.1$ for pyramidal regularization. Pyramidal regularization fills all occlusion holes allowing movement away from the original viewpoint samples. Smooth hole filling also makes the texture easier to compress by removing hard boundaries between defined and undefined samples. Regularization makes MIPMAP levels tend toward filtered versions of each other; we exploit this by compressing only the finest level and re-creating higher levels by on-the-fly decimation in the decoder.

The top row of Figure 3.8 shows results for the “forward mapping” method in which texture samples are mapped to the object’s image layer and interpolated using a high-quality filter. We used a separable Lanczos-windowed sinc function with a halfwidth of 4. To handle occlusions, we first filled undefined samples in the segmented layer using a simple boundary-reflection algorithm. Forward mapping produces a blurry and inaccurate result because it does not account for how graphics hardware filters the textures. In addition, reflection hole-filling produces artificial, high-frequency information in occluded regions that is expensive to encode.

Finally, the bottom row of Figure 3.8 shows results for the method of ray-tracing directly into texture maps. We generated texture maps for this method by rendering directly into textures that are supersampled by a factor 3, and then down-sampled the textures to the actual resolution using a separable Lanczos-windowed sinc decimation filter with a halfwidth of 6. As shown, ray-tracing directly into texture maps also produces a blurry and inaccurate result because it does not account for how graphics hardware filters the textures. This effect is more easily seen in Figure 3.10 and Figure 3.11. Again, there is the problem of filling in occluded regions, and the associated problem of how to build MIPMAPs. In our solution, we ignore whether a surface happens to be front or back-facing which simplifies building of MIPMAPs by mimicking reflection hole-filling, but places artificial, high-frequency information in occluded regions that is expensive to encode.

Figure 3.10 and Figure 3.11 compare inference methods for a bilinear and anisotropic hardware filter, respectively. It is more significant to compare the results for the anisotropic filter because this filter more robustly handles changes in viewpoint and resolution. In terms of peak signal to noise ratio, there is a 2.4 dB difference between the least-squares method and the forward mapping method, and a 3.4 dB difference between least-squares method and the method of ray-traced texture maps. We conclude that by considering the hardware filter model that is used when texture mapping geometry in graphics hardware, and by using a least-squares optimization approach, we can produce images on graphics hardware that more accurately match the original ray-traced images than alternative approaches.

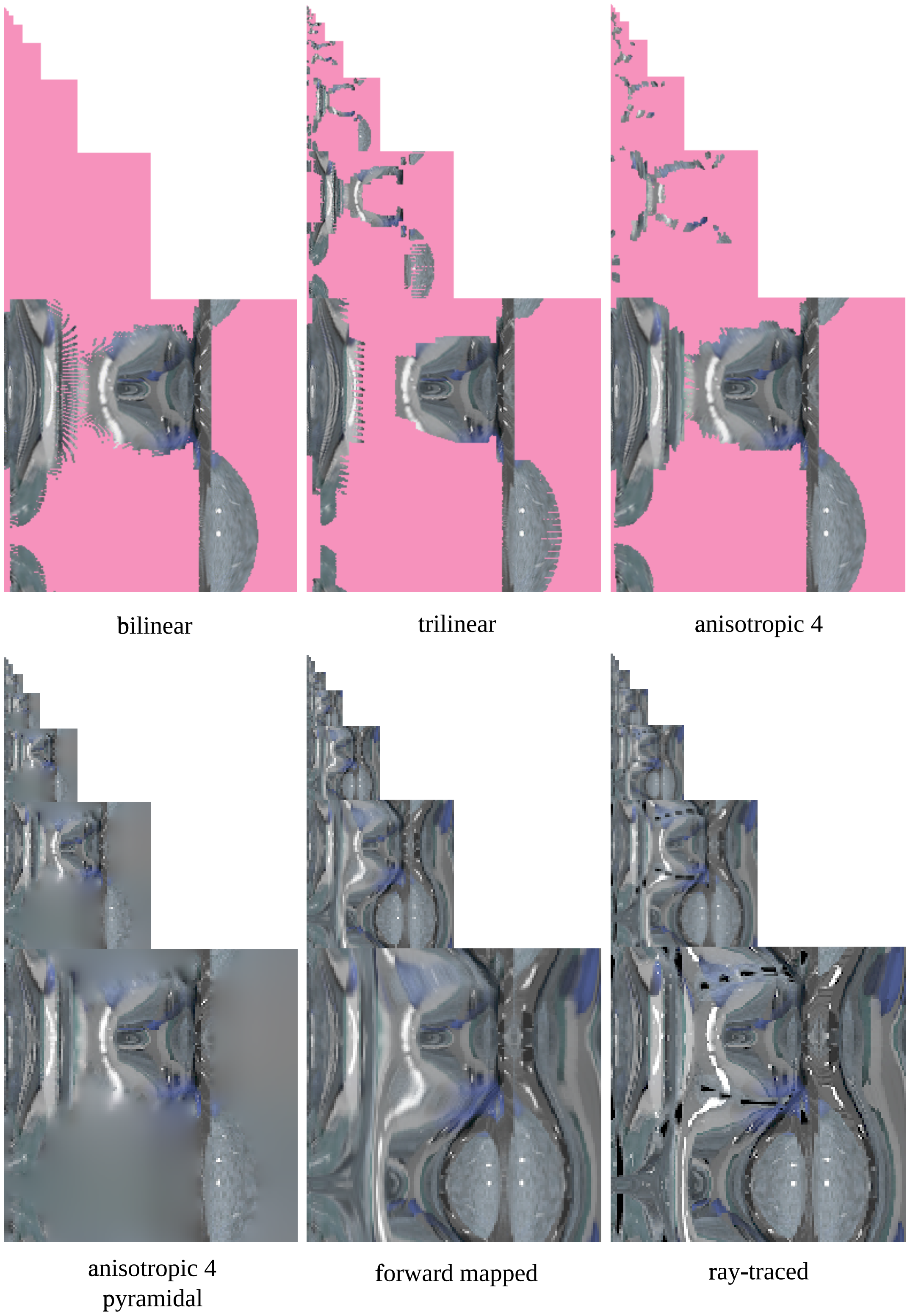
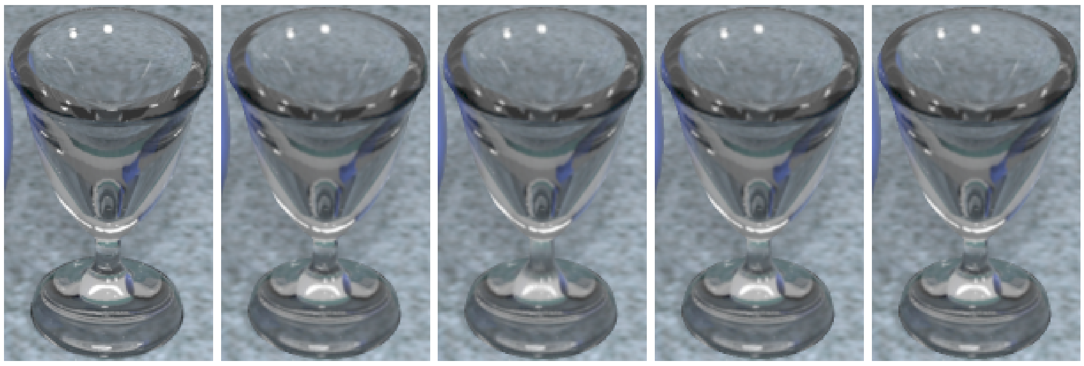
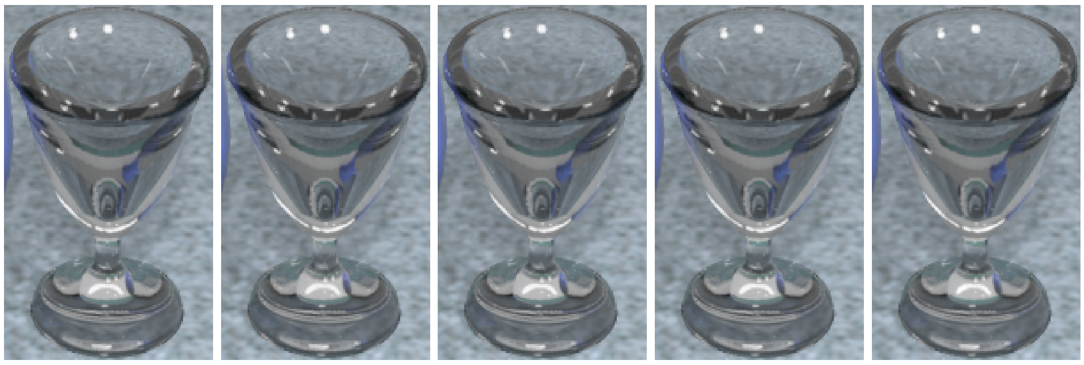


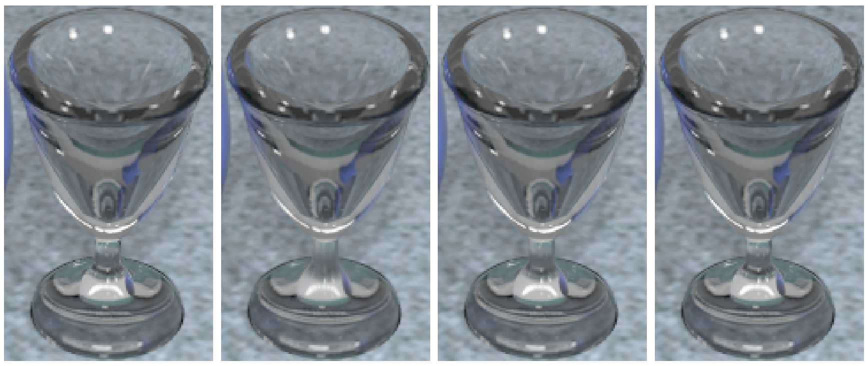
Figure 3.7: **Inferred Texture Maps for the Cup Object.** Pink regions represent undefined regions of the texture. The pyramidal texture is inferred with $\varepsilon_f = 0.1$.



	Forward Mapping Method			
Original	bilinear	trilinear	anisotropic 2	anisotropic 4
	MSE=13.4 PSNR=36.9 dB	MSE=17.7 PSNR=35.6 dB	MSE=13.6 PSNR=36.8 dB	MSE=12.8 PSNR=37.0 dB

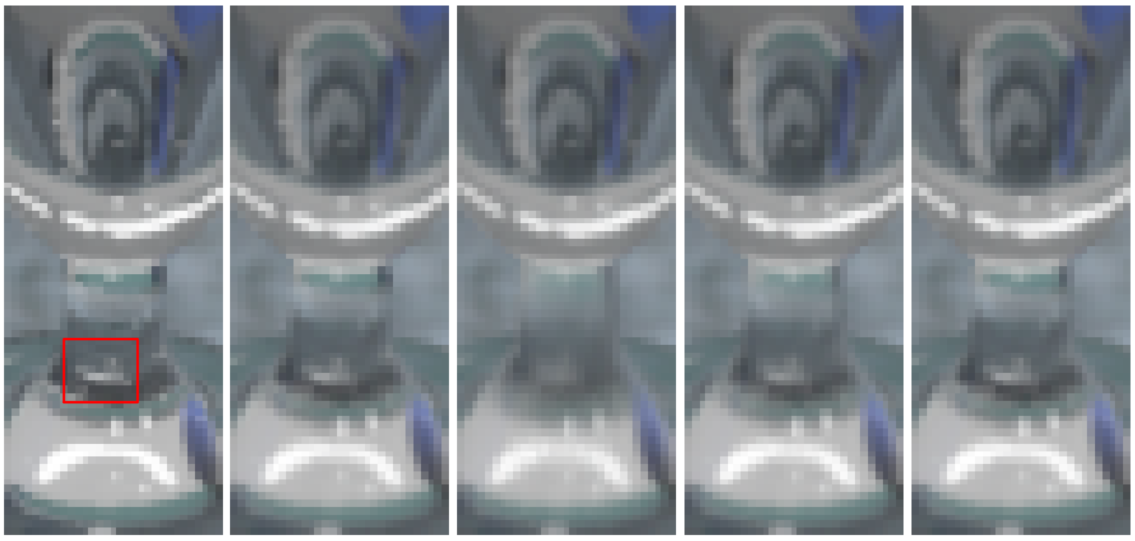


Least-Squares Method				
bilinear	trilinear	anisotropic 2	anisotropic 4	pyramidal
MSE=7.24 PSNR=39.5 dB	MSE=10.3 PSNR=38.0 dB	MSE=7.94 PSNR=39.1 dB	MSE=7.49 PSNR=39.4 dB	MSE=7.21 PSNR=39.6 dB



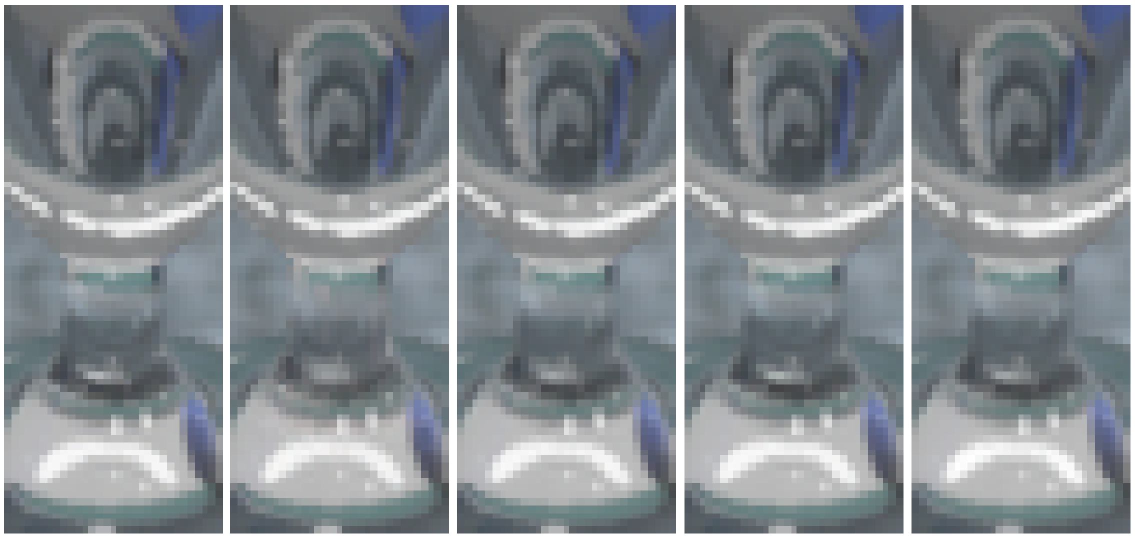
Ray-Traced Texture Maps			
bilinear	trilinear	anisotropic 2	anisotropic 4
MSE=17.1 PSNR=35.8 dB	MSE=20.2 PSNR=35.1 dB	MSE=16.9 PSNR=35.8 dB	MSE=16.2 PSNR=36.0 dB

Figure 3.8: **Image comparison across texture filtering modes and inference methods.** The pyramidal image is generated with an anisotropy factor of 4.



	Forward Mapping Method			
Original	bilinear	trilinear	anisotropic 2	anisotropic 4

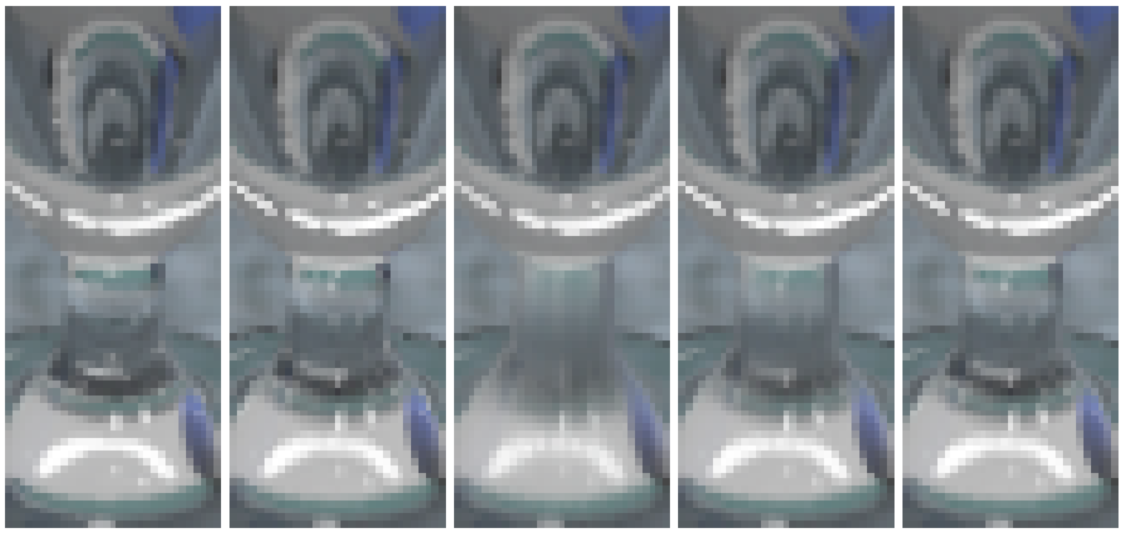
	MSE=19.1
	PSNR=35.3 dB
		MSE=55.5
		PSNR=30.7 dB
			MSE=27.7
			PSNR=33.7 dB
				MSE=20.3
				PSNR=35.1 dB



Least-Squares Method				
bilinear	trilinear	anisotropic 2	anisotropic 4	pyramidal

MSE=6.76
PSNR=39.8 dB
	MSE=28.1
	PSNR=33.6 dB
		MSE=11.6
		PSNR=37.5 dB
			MSE=8.36
			PSNR=38.9 dB
				MSE=8.67
				PSNR=38.8 dB

Figure 3.9: **Close-up image comparison.** The highlight within the red box is a good place to observe differences.



	Ray-Traced Texture Maps			
Original	bilinear	trilinear	anisotropic 2	anisotropic 4

	MSE=32.7
	PSNR=33.0 dB
		MSE=68.1
		PSNR=29.8 dB
			MSE=35.8
			PSNR=32.6 dB
				MSE=28.6
				PSNR=33.6 dB

Figure 3.9: **Close-up image comparison (continued).**

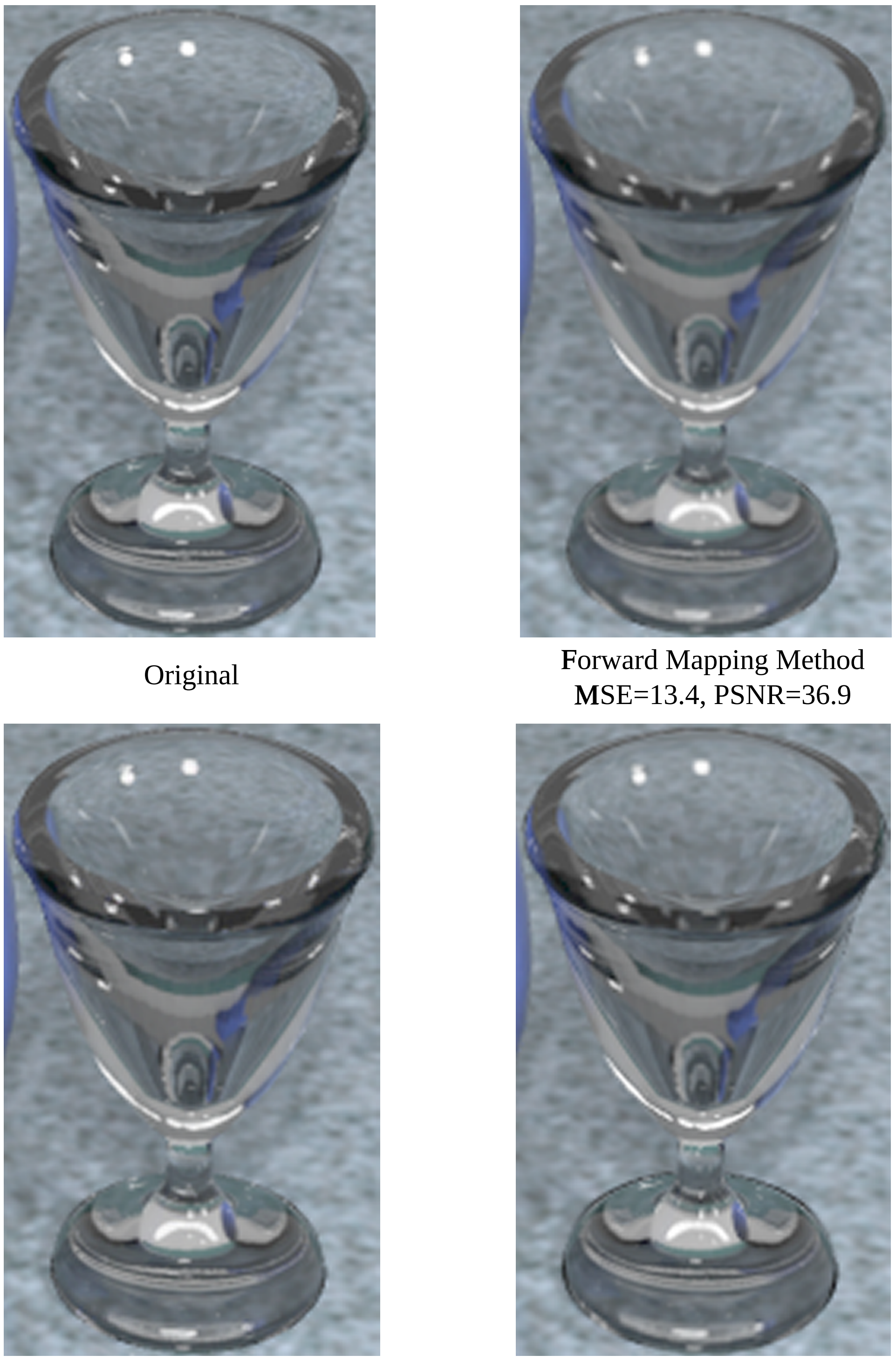
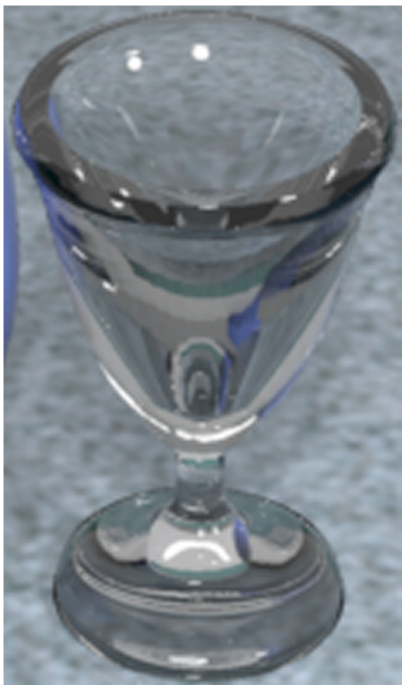
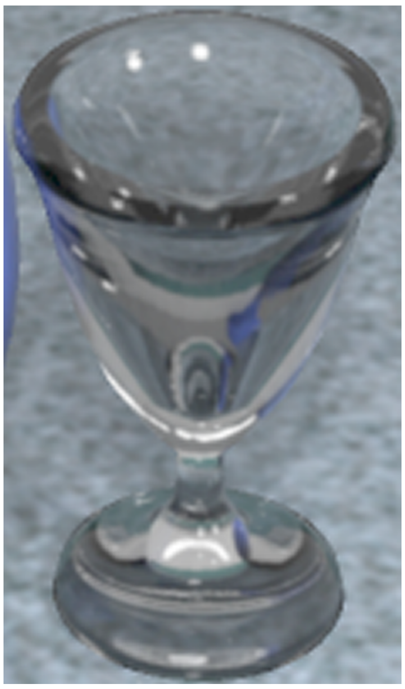
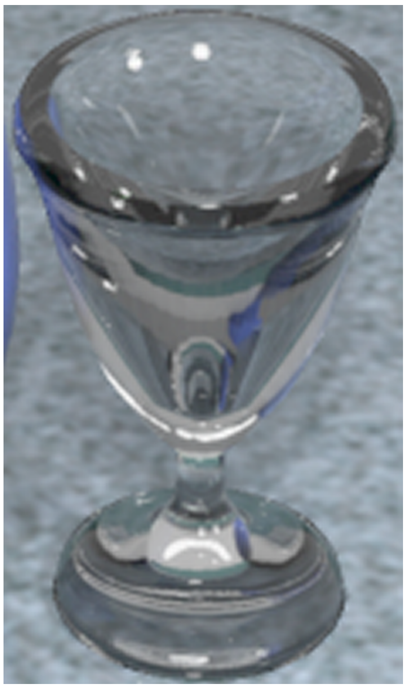


Figure 3.10: **Image comparison across inference methods with bilinear texture filtering.**


Original


Forward Mapping Method
MSE=12.8, PSNR=37.0


Least-Squares Method
MSE=7.49, PSNR=39.4

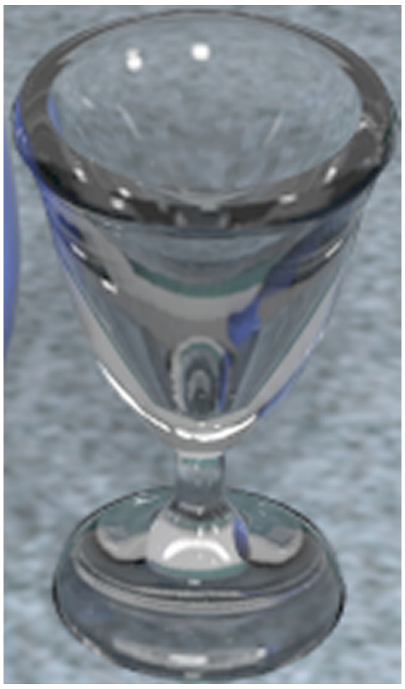

Ray-Traced Texture Maps
MSE=16.2, PSNR=36.0

Figure 3.11: **Image comparison across inference methods with anisotropic filtering.** The images were rendered with hardware that supports an anisotropy factor up to 4.

Chapter 4

Parameterized Texture Compression

For synthetic scenes, the time and viewpoint parameters of the plenoptic function [Ade91, McM95] can be generalized. We are free to parameterize the radiance field based on time, position of lights or viewpoint, surface reflectance properties, object positions, or any other degrees of freedom in the scene, resulting in an arbitrary-dimensional parameterized animation. Our goal is maximum compression of the parameterized animation that maintains satisfactory quality and decodes in real time. Once the encoding is downloaded over a network, the decoder can take advantage of specialized hardware and high bandwidth to the graphics system allowing a user to explore the parameter space. High compression reduces downloading time over the network and conserves server and client storage.

Figure 4.1 illustrates our system. Ray-traced images at each point in the parameter space are fed to the compiler together with the scene geometry, lighting models, and viewing parameters. The compiler targets any desired type of graphics hardware and infers texture resolution, texture domain mapping, and texture samples for each object over the parameter space to produce as good a match as possible on that hardware to the "gold-standard" images (as discussed in Chapter 3). Per-object texture maps are then compressed using a novel, multi-dimensional compression scheme. The interactive runtime consists of two parts operating simultaneously: a texture decompression engine and a traditional hardware-accelerated rendering engine.

The contributions of this chapter are as follows:

- We introduce the problem of compressing multidimensional animations, not just radiance fields parameterized by viewpoint or animations through 1D time.

- We fully exploit cheap and ubiquitous graphics hardware by rendering texture maps on geometric objects rather than view-based images. We employ an automatic method to allocate storage over objects texture maps and select texture map resolutions and domains based on

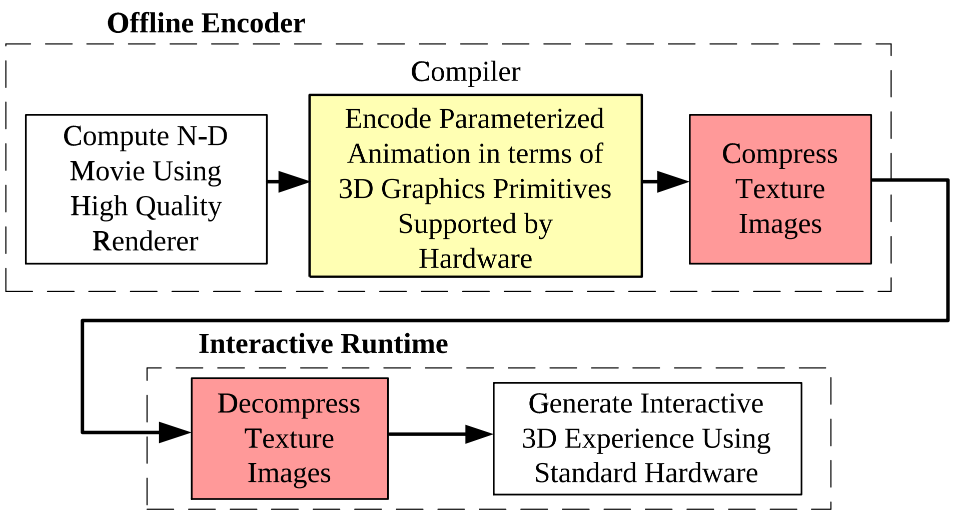

Offline Encoder
Compiler
Compute N-D Movie Using High Quality Renderer
Encode Parameterized Animation in terms of 3D Graphics Primitives Supported by Hardware
Compress Texture Images
Interactive Runtime
Decompress Texture Images
Generate Interactive 3D Experience Using Standard Hardware

Figure 4.1: **System Overview.**

the gold-standard images. We also separate diffuse and specular lighting layers to increase compression, using automatic storage allocation over these lighting layers. (Section 4.5)

- We present novel methods for general, multidimensional compression using an adaptive Laplacian pyramid that allows real-time decoding and high compression ratios. (Section 4.3)

- We describe a novel run-time system that caches to speed texture decoding and staggers block origins to distribute decompression load. (Section 4.7)

- We present realistic, highly specular examples with multiple objects containing thousands of polygons, using a PC equipped with a consumer graphics card. The quality and generality of our examples exceed previous work in image-based rendering. We demonstrate the superiority of our encoding over alternatives like MPEG4 and show high-quality results at compression ratios of 200-800 with near real-time (˜5Hz) decoders capable of hardware implementation. Faster decoding (˜31Hz) is also possible at reduced quality. Since the systems main bottleneck is texture decompression, our findings provide incentive for incorporating more sophisticated texture decompression functionality in future graphics pipelines. (Section 4.8)

The limitations of our approach are as follows:

- We assume that a list of the geometric objects and their texture parameterizations are given as input.

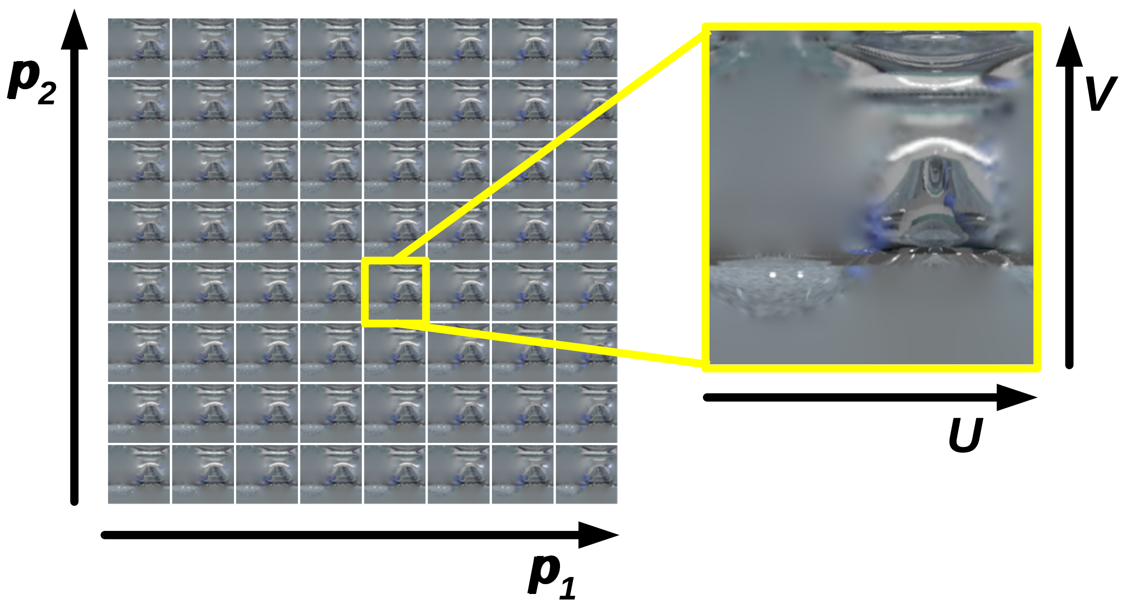


Figure 4.2: **Parameter Block Compression.** An 8x8 block of parameterized textures for a glass cup object is shown. In this example, dimension p1 represents a 1D viewpoint trajectory while p2 represents the swinging of a light source. Note the high degree of coherence in the texture maps. One of the textures is shown enlarged, parameterized by the usual spatial parameters, denoted u and v. We use a Laplacian pyramid to encode the parameter space and standard 2D compression such as block-based DCT to further exploit spatial coherence within each texture (i.e., in u and v).

- Efficient encoding relies on parameter-independent geometry; that is, geometry that remains static or rigidly moving and thus represents a small fraction of the storage compared to the parameter-dependent textures. For each object, polygonal meshes with texture coordinates are encoded once as header information.

- The compiler must have access to an image at each point in parameter space, so compilation is exponential in dimension. We believe our compilation approach is good for spaces in which all but one or two dimensions are secondary; i.e., having relatively few samples. Examples include viewpoint movement along a 1D trajectory with limited side-to-side movement, viewpoint changes with limited, periodic motion of some scene components, or time or viewpoint changes coupled with limited changes to the lighting environment.

4.1 Parameter Space Blocks

The multidimensional field of textures for each object is compressed by subdividing into parameter space blocks as shown in Figure 4.2. Larger block sizes better exploit coherence but are more costly to decode during playback; we used 8×8 blocks in our 2D examples.

4.2 Why compress parameterized textures instead of parameterized images?

Our approach infers and compresses parameter-dependent texture maps for individual objects rather than combined views of the entire scene. Encoding a separate texture map for each object has several advantages:

- Textures better capture coherence across the parameter space independently of where in the image an object appears. This is particularly the case for diffusely shaded objects.
- Object silhouettes are correctly rendered from actual geometry and suffer fewer compression artifacts.
- The viewpoint can move from the original parameter samples without revealing geometric disocclusions.

4.3 Adaptive Laplacian Pyramid

We encode parameterized texture blocks using a Laplacian pyramid [Bur83]. Consider a single (u, v) texture sample, parameterized by a *d*-dimensional space $\{p_1, p_2, \ldots, p_d\}$ with n samples in each dimension of the block. Starting from the finest (bottom) level with n^d samples, the parameter samples are filtered using a Gaussian kernel and subsampled to produce coarser versions, until the top of the pyramid is reached containing a single sample that averages across all of parameter space. Each level of the pyramid represents the detail that must be added to the sum of the higher levels in order to reconstruct the signal. Coherent signals have relatively little information at the lower levels of the pyramid, so this structure supports efficient encoding.

Though the Laplacian pyramid is not a critically sampled representation, it requires just $\log_2(n)$ simple image additions in order to reconstruct a leaf image. In comparison, a multidimensional Haar wavelet transform requires $(2^d - 1)\log_2(n)$ image additions and subtractions. Another advantage of the Laplacian Pyramid is that graphics hardware can perform the necessary image additions using multiple texture stages, thus enabling on-the-fly decompression. For decoding speed, we reconstruct

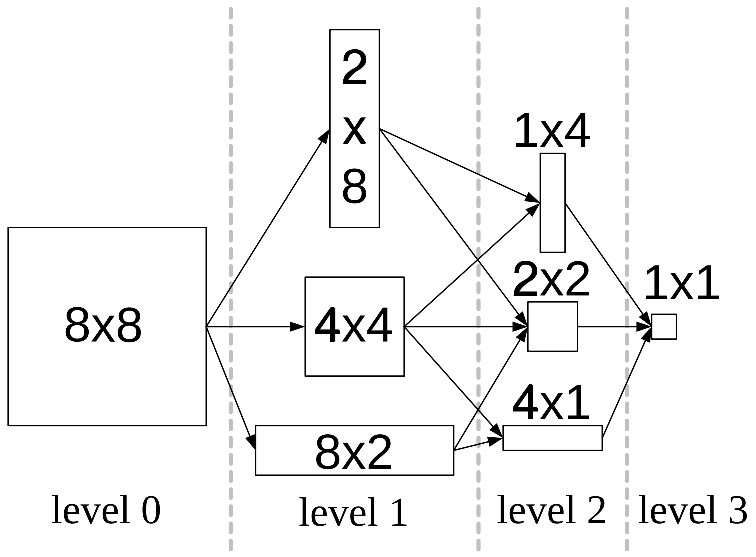


Figure 4.3: **Adaptive Laplacian Pyramid.**

using the nearest-neighbor parameter sample; higher-order interpolation temporally smooths results but is much more expensive.

The "samples" at each pyramid level are entire 2D images rather than samples at a single (u, v) location. We use standard 2D compression (e.g., JPEG [Pen92, Xio96] and SPIHT [Sai96] encodings) to exploit spatial coherence over (u, v) space. Each level of the Laplacian pyramid thus consists of a series of encoded 2D images. Parameter and texture dimensions are treated asymmetrically because parameters are accessed along an unpredictable 1D subspace selected by the user at run-time. We can not afford to process large fractions of the representation to decode a given parameter sample, a problem solved by using the Laplacian pyramid with fairly small block size.

In contrast, texture maps are atomically decoded and loaded into the hardware memory and so provide more opportunity for a software codec that seeks maximum compression without regard for random access. We anticipate that texture map decoding functionality will soon be absorbed into graphics hardware [Bee96]. In that case, whatever compressed representation the hardware consumes is a good choice for the leaf node texture maps.

It is typically assumed in image coding that both image dimensions are equally coherent. This assumption is less true of parameterized animations where, for example, the information content in a viewpoint change can greatly differ from that of a light source motion. To take advantage of differences in coherence across different dimensions, we use an *adaptive* Laplacian pyramid that

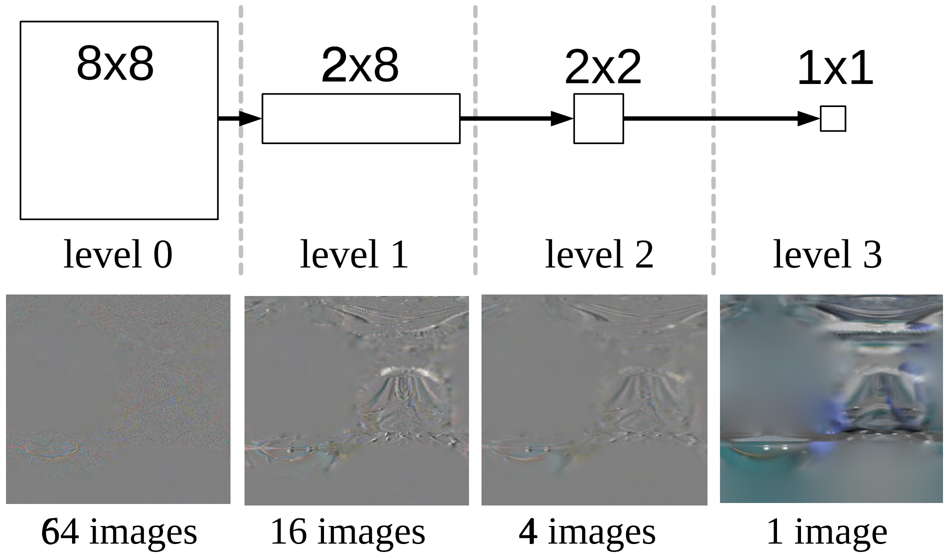


Figure 4.4: **Example Images from each Level of an Adaptive Laplacian Pyramid.**

subdivides more in dimensions with less coherence. Figure 4.3 illustrates all the possible permutations of a 2D adaptive pyramid with four levels, in which coarser levels still have 4 times fewer samples as in the standard Laplacian pyramid. Though not shown in the figure, it is also possible to construct pyramids with different numbers of levels, for example to "jump" directly from an 8×8 level to an 8×1. We pick the permutation that leads to the best compression using a greedy search. Figure 4.4 illustrates texture images at each level of an adaptive Laplacian pyramid.

4.4 MPEG Encoding of Parameterized Textures

As mentioned above, the main reasons for picking the Laplacian pyramid approach is that it provides high compression by exploiting multi-dimensional coherence, and at the same time supports fast decoding by limiting processing to just one image from each level of the pyramid. One alternative approach is MPEG encoding of parameterized textures.

The idea is to perform MPEG encoding in a 2D or higher-dimensional space by taking a zig-zag path through the space varying the dimension of most coherence most rapidly. One of the parameters of MPEG encoding is the spacing between I-frames, or number of I-frames per block. Having a single I-frame/block maximizes compression, but increases decoding time. In the worst case, accessing a parameterized texture requires 23 inverse DCT operations, 22 forward predictions, 1 backward prediction and 1 interpolation prediction for the single I/block case. We do not believe the 1I/block encoding is practical for real-time decoding. This analysis is also true for when MPEG is used to encode parameterized images instead of textures. Note that decreasing the number of I-frames per block in MPEG is somewhat analogous to increasing the block size, and thus the number of levels, in our pyramid schemes – both trade-off decoding speed for better compression. At 10 I-frames/block, 4 inverse DCT's, 2 forward predictions, 1 backward prediction, and 1 interpolative prediction are required in the worst case. This is roughly comparable to our DCT Laplacian pyramid decoding, which also requires 4 inverse DCT operations, though pyramid reconstruction involves only 3 image additions rather than more complicated motion predictions. Results for MPEG encoding of textures are presented in Section 4.8 and compared against our Laplacian pyramid approach.

4.5 Automatic Storage Allocation

To encode the Laplacian pyramid, storage must be assigned to its various levels. We apply standard bit allocation techniques from signal compression [Ger92, p.606-610]. Curves of mean squared error versus storage, called *rate/distortion curves*, are plotted for each pyramid level and points of equal slope on each curve selected subject to a total storage constraint. More precisely, let $\overline{E}_i(r_i)$ be the mean squared error (MSE) in the encoding of level i when using r_i bits. It can be shown that the minimum sum of MSE over all levels subject to a total storage constraint of R; i.e.,

$$\min \sum_i \overline{E}_i(r_i) \ \ni \ \sum_i r_i = R$$

occurs when the $\overline{E}_1' = \overline{E}_2' = \cdots = \overline{E}_m'$, where m is the total number of levels and $\overline{E}_i' = d\overline{E}_i/dr_i$. We minimize the sum of MSEs because a texture image at a given point in parameter space is reconstructed as a sum of images from each level, so an error in any level contributes equally to the resulting error. A simple 1D root finder suffices to find $\overline{E}_i'$ from which the r_i can be derived by inverting the rate/distortion curve at level i.

There is also a need to perform storage allocation across objects; that is, to decide how much to spend in the encoding of object A's texture vs. object B's. This kind of storage allocation is needed for both the Laplacian pyramid and MPEG approaches of encoding textures. We use the same method as for allocating between pyramid levels, except that the error measure is $E_i \equiv A_i\overline{E}_i$, where A_i is the screen area and $\overline{E}_i$ the MSE of object i. This minimizes the sum of squared errors on the screen no matter how the screen area is decomposed into objects. To speed processing, we compute errors in texture space rather than rendering the textures and computing image errors. We find this provides an acceptable approximation.

A complication that arises is that there can be large variations in MSE among different objects,

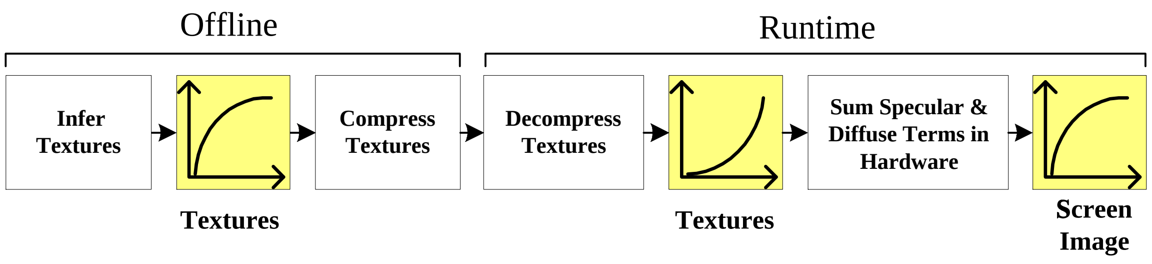


Figure 4.5: **Compensation for Gamma Correction.**

some of which can be perceptually important foreground elements. We therefore introduce a constraint that any objects MSE satisfy $\overline{E}_i \leq \alpha\overline{E}$ where $\overline{E}$ is the average MSE of all objects and $\alpha > 1$ is a user-specified constant. A two-pass algorithm is used in which we first minimize $\sum_i E_i$ over objects subject to an overall storage constraint. Using the resulting $\overline{E}$, we then eliminate the part of the rate distortion curves of any object that incurs more MSE than $\alpha\overline{E}$ and solve again. This reallocates storage from objects with low MSEs to objects with above-threshold MSEs in such a way as to minimize sum of squared error in the below-threshold objects.

The above algorithms can also be used as a starting point for manual allocation of storage across objects, so that more important objects can be more faithfully encoded.

For objects with both specular and diffuse reflectance, we encode separate lighting layers for which storage must be allocated. We use the method described above on the entire collection of textures across objects and lighting layers.

4.6 Compensation for Gamma Correction

Splitting an object's lighting layers into the sum of two terms conflicts with gamma correction, since $\gamma(L_1 + L_2) \neq \gamma(L_1) + \gamma(L_2)$ where L_i are the lighting layers and $\gamma(x) = x^{1/g}$ is the (nonlinear) gamma correction function. Typically, $g \approx 2.2$. Without splitting, we can simply match texture maps to a gamma-corrected version of the gold standard, although this is not entirely correct because hardware would be linear filtering the gamma-corrected texture. With splitting, we instead infer textures from the original, uncorrected layers so that sums are correctly performed in a linear space, and gamma correct as a final step in the hardware rendering. The problem arises because gamma correction magnifies compression errors in the dark regions.

To compensate, we instead encode based on the gamma corrected signals, $\gamma(L_i)$, effectively scaling up the penalty for compression errors in the dark regions. At run-time, we apply the inverse gamma correction function $\gamma^{-1}(x) = x^g$ to the decoded result before loading the texture into

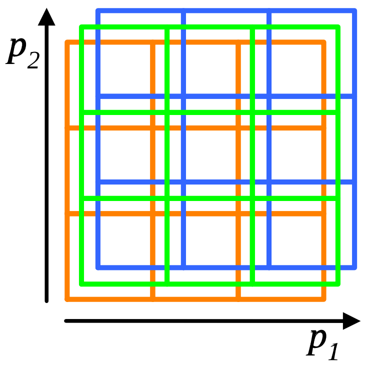


Figure 4.6: **Staggering of block origins.** We use different block origins for different objects.

hardware memory, and, as before, sum using texture operations in a linear space and gamma correct the final result. This process is illustrated in Figure 4.5. We note that the inverse gamma function employed, as well as gamma correction at higher precision than the 8-bit framebuffer result, is a useful companion to hardware decompression.

4.7 Runtime System

The runtime system decompresses and caches texture images, applies affine transformations to vertex texture coordinates, and generates rendering calls to the graphics system. Movement off (or between) the original viewpoint samples is allowed simply by rendering from that viewpoint using the closest texture sample.

The texture caching system decides which textures to keep in memory in decompressed form. Because the user's path through parameter space is unpredictable, we use an adaptive caching strategy based on the notion of lifetimes. Whenever a texture image is accessed, we reset a count of the number of frames since the image was last used. When the counter exceeds a given lifetime, L, the memory for the decompressed image is reclaimed. Different levels of the Laplacian pyramid have different levels of priority since images near the top are more likely to be reused. Lifetimes are therefore computed as $L = ab^{\,l}$ where a is a constant that represents the lifetime for leaf nodes (typically 20), b is the factor of lifetime increase for higher pyramid levels (typically 4) and l represents pyramid level. Note that the number of images cached at each pyramid level and parameter space block changes dynamically in response to user behavior.

If blocks of all objects are aligned, then many simultaneous cache misses occur whenever the user crosses a block boundary, creating a computational spike as multiple levels in the new blocks'

Laplacian pyramids are decoded. We mitigate this problem by *staggering* the blocks, as illustrated in Figure 4.6, using different block origins for different objects, to more evenly distribute decompression load.

4.8 Compression and Performance Results

4.8.1 Demo1: Light $\times$ View

4.8.1.1 Compression Results

The first example scene (Figure 4.9, top left) consists of 6 static objects: a reflective vase, glass cup, reflective table top, table stand, walls, and floor. It contains 4384 triangles and was rendered in about 5 hours/frame on a group of 400Mhz Pentium II PCs, producing gold standard images at 640×480 resolution. The 2D parameter space has 64 viewpoint samples circling around the table at 1.8°/sample and 8 different positions of a swinging, spherical light source[1]. The image field was encoded using eight 8×8 parameter space blocks, each requiring storage 640×480×3×8×8= 56.25MB/block.

Our least-squares texture inference method created parameterized textures for each object, assuming trilinear texture filtering. The resulting texture fields were compressed using a variety of methods, including adaptive 2D Laplacian pyramids of both DCT- and SPIHT-encoded levels. Storage allocation over objects was computed using the method of Section 4.5, with a max MSE variation constraint of $\alpha = 1.25$. The decoded textures were then applied in a hardware rendering on a graphics card with the Nvidia Geforce 256 chip, 32MB local video memory, and 16MB nonlocal AGP memory running on a Pentium III 733Mhz PC. To test the benefits of the Laplacian pyramid, we also tried encoding each block using MPEG on a 1D zig-zag path through the parameter space varying most rapidly along the dimension of most coherence (Figure 4.7). A state-of-the-art MPEG4 encoder [MIC99] was used. Finally, we compared against direct compression of the original images (rather than renderings using compressed textures), again using MPEG 4 with one I-frame per block. This gives MPEG the greatest opportunity to exploit coherence with motion prediction.

Figures 4.9 and 4.10 show the results at a targeted compression rate of 384:1 and Figures 4.11 and 4.12 shows the results at a targeted compression rate of 768:1, representing target storage of 150 KB/block and 75 KB/block respectively. Due to encoding constraints, some compression ratios undershot the target and are highlighted in yellow. All texture-based images were generated on graphics hardware using 2×2 antialiasing; their MSEs were computed from the framebuffer contents. MSEs were averaged over an entire block of parameter space.

[1]The specific parameters were light source radius of 8, pendulum length of 5, distance of pendulum center from table center of 71, and angular pendulum sweep of 22°/sample.

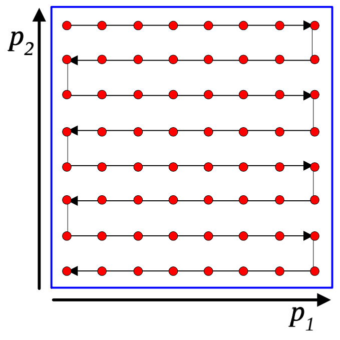


Figure 4.7: **MPEG 1D zig-zag path through a single** 8×8 **block of parameter space.** The zig-zag path is chosen so that the dimension of most coherence varies most rapidly, in this example dimension p_1.

Both Laplacian pyramid texture encodings (right two columns, bottom row) achieve reasonable quality at 768:1, and quite good quality at 384:1. The view-based MPEG encoding, labeled "MPEG-view", is inferior with obvious block artifacts on object silhouettes, even though MPEG encoding constraints did not allow as much compression as the other examples. The SPIHT-encoded Laplacian pyramid is slightly better than DCT, exhibiting blurriness rather than block artifacts (observe the left hand side of the vase for the 768:1 rate). The differences in the pyramid schemes between the 384:1 and 768:1 targets are fairly subtle, but can be seen most clearly in the transmitted image of the table top through the glass cup. Of course, artifacts visible in a still image are typically much more obvious temporally.

For MPEG encoding of textures we tried two schemes: one using a single I-frame per block (IBBPBBP. . .BBP) labeled "MPEG-texture 1I-frame/block", and another using 10 I-frames (IBBPBBIBBPBBI. . .IBBP) labeled "MPEG-texture 10I-frames/block". The zig-zag path was chosen so that the dimension of most coherence varies most rapidly, in this case the light position dimension (Figure 4.7). As mentioned in Section 4.4, though single I-frame/block maximizes compression, it does so at the cost of decoding time. We do not believe the 1 I-frame/block encoding is practical for real-time decoding, but include the result for quality comparison. MPEG encoding with 10 I-frames per block is roughly comparable to our DCT Laplacian pyramid decoding.

The 10 I-frame/block MPEG-texture results have obvious block artifacts at both quality levels especially on the vase and green wallpaper in the background. They are inferior to the pyramid encodings. This is true even though we were unable to encode the scene with higher compression than 418:1, significantly less than the other examples at the 768:1 target rate. This result is not surprising given that MPEG can only exploit coherence in one dimension. The 1 I-frame/block results are better, but still inferior to the pyramid schemes at the 384:1 target, where the vase exhibits noticeable block artifacts. For the 768:1 target, the quality of MPEG-texture 1 I-frame/block falls between the SPIHT and DCT pyramids. Unlike the MPEG-view case, the MPEG-texture schemes still use many of the novel features of our approach: hardware-targeted texture inference, separation of lighting layers, and optimal storage allocation across objects.

Figure 4.13 isolates the benefits of lighting separation and adaptive Laplacian subdivision. These results were achieved with the Laplacian SPIHT encoding at the 384:1 target. With combined lighting layers, adaptive subdivision increases fidelity especially noticeable in the table seen through the glass cup (Figures 4.13a and b); MSE across the block is reduced 20%. This is because textures, especially the glass cup's, change much less over the light position dimension than over the view dimension. In response, the first level of pyramid subdivision occurs entirely over the view dimension. We then separately encode the diffuse and specular lighting layers, still using adaptive subdivision (Figure 4.13c). While this increases MSE slightly because additional texture layers must be encoded[2], the result is perceptually better, producing sharper highlights on the vase.

4.8.1.2 System Performance

Average compilation and preprocessing time per point in parameter space is shown in Table 4.1. It can be seen that total compilation time is a small fraction of the time to produce the ray-traced images.

texture coordinate optimization	1 sec
obtaining matrix A	2.15 min
solving for textures	2.68 min
storage allocation across pyramid levels	0.5 min
storage allocation across objects	1 sec
compression	5 sec
total compilation	5.43 min
ray tracing	5 hours

Table 4.1: Parameterized Texture Map Compilation Time per Image in Parameter Space

To determine playback performance, we measured average and worst-case frame rates (fps) for a diagonal trajectory that visits a separate parameter sample at every frame. The results for both

[2]Only the table-top and vase objects had separately encoded diffuse and specular layers; they were the only objects with diffuse and reflective terms in their shading model. Thus a total of 8 parameterized textures were encoded for this scene.

DCT- and SPIHT-Laplacian pyramid encodings are summarized in Table 4.2 and used compression at the 384:1 target.

Encoding	**Texture**	**Worst fps**	**Average fps**
Laplacian DCT	undecimated	2.46	4.76
	decimated	18.4	30.7
Laplacian SPIHT	undecimated	0.27	0.67
	decimated	2.50	5.48

Table 4.2: Parameterized Texture Map Runtime Performance

The performance bottleneck is currently software decoding speed. To improve performance, we tried encoding textures at reduced resolution. Reducing texture resolution by an average factor of 11 (91%) using a manually specified reduction factor per object provides acceptable quality at about 31 fps with DCT. Decoding speedup is not commensurate with resolution reduction because it partially depends on signal coherence and decimated signals are less coherent.

4.8.2 Demo2: View $\times$ Object Rotation

In the second example, we added a rotating, reflective gewgaw on the table. The parameter space consists of a 1D circular viewpoint path, containing 24 samples at 1.5°/sample, and the rotation angle of the gewgaw, containing 48 samples at 7.5°/sample. Results are shown in Figure 4.14 for encodings using MPEG-view and Laplacian SPIHT.

This is a challenging example for our method. There are many specular objects in the scene, reducing the effectiveness of lighting separation (the gewgaw and glass cup have no diffuse layer). The parameter space is much more coherent in the rotation dimension than in the view dimension, because gewgaw rotation only changes the relatively small reflected or refracted image of the gewgaw in the other objects. On the other hand, the gewgaw itself is more coherent in the view dimension because it rotates faster than the view changes. MPEG can exploit this coherence very effectively using motion compensation along the rotation dimension. Though our method is designed to exploit multidimensional coherence and lacks motion compensation, our adaptive pyramid also responds to the unbalanced coherence, producing a slightly better MSE and a perceptually better image.

To produce these results, we manually adjusted the storage allocation over objects. Shading on the background objects (walls, floor, and table stand) is static since they are diffuse and the gewgaw casts no shadows on them. Their storage can thus be amortized over all 18 blocks of the parameter space. Because they project to a significant fraction of the image and can be so efficiently compressed, our automatic method gives them more storage than their perceptual importance warrants.

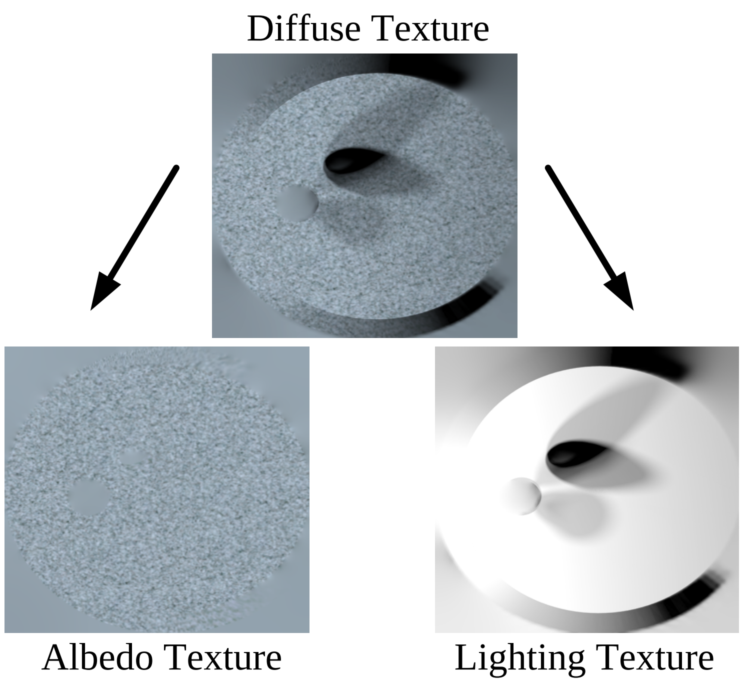


Figure 4.8: **Separation of Diffuse Texture into Albedo and Lighting Textures.**

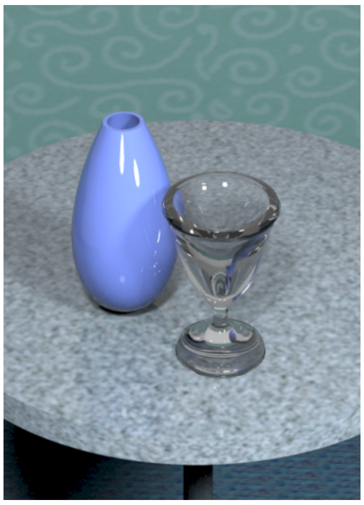
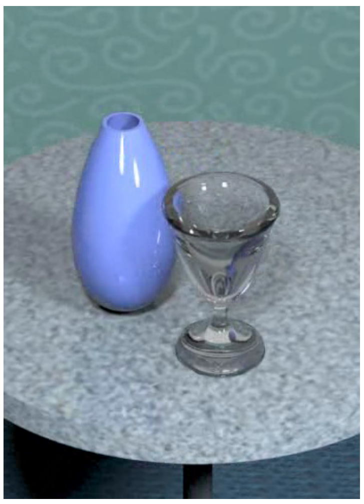
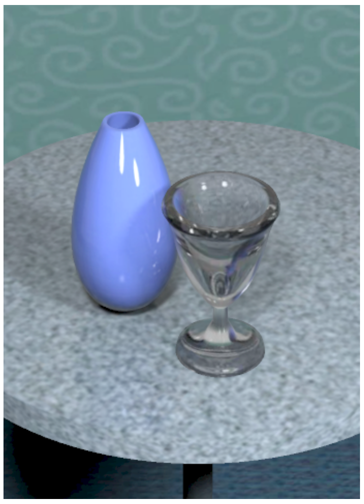
We reduced their allocation by 72% and devoted the remainder to an automatic allocation over the foreground objects. Even with this reduction, the texture-based encoding produces less error on the background objects, as can be seen in Figure 4.14.

Real-time performance for this demo is approximately the same as for demo1.

4.9 Discussion

Our simple sum of diffuse and specular texture maps is but a first step toward more predictive graphics models supported by hardware to aid compression. One example optimization is to take advantage of texture multiplication in graphics hardware and further segment the diffuse layer into separate albedo and lighting layers. This segmentation, illustrated in Figure 4.8, is useful for separating a high-frequency but parameter-independent albedo map from a low-frequency, parameter-dependent incident irradiance field. The albedo map need only be compressed once with the storage cost amortized over all blocks in the space. Other examples of graphics models that may improve compression include parameterized environment maps to encode reflections (as discussed in the next Chapter), hardware shadowing algorithms, bump-mapping hardware support, and per-vertex shading models.

With respect to run-time performance, our present system decompresses and loads into hardware memory entire textures rather than simply loading regions of textures that are actually accessed during rendering. The consequence of this simple approach is that some fraction of the decoding computation and memory bandwidth are wasted. This inefficiency can be reduced by taking advantage of information available about texture accesses during compilation to keep a record of which regions of the texture to decode and load at run-time, albeit at a slight increase in storage cost. Alternatively, a more elegant solution is for graphics hardware to adopt a "virtual memory" system for texture maps, where only regions of the texture that are actually accessed during rendering are decoded and *paged* into memory.

	View-Based	Texture-Based
Original	MPEG-view 1 I-frame/block	MPEG-texture 1 I-frame/block
		
	355:1 MSE=13.5, PSNR=36.8 dB	384:1 MSE=10.1, PSNR=38.1 dB

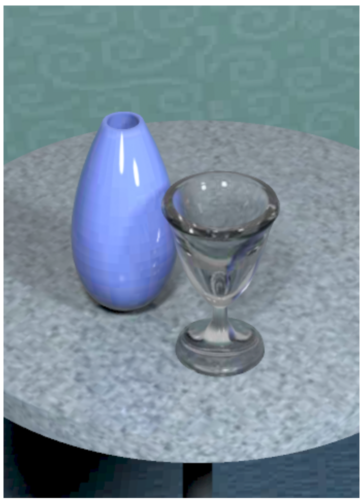
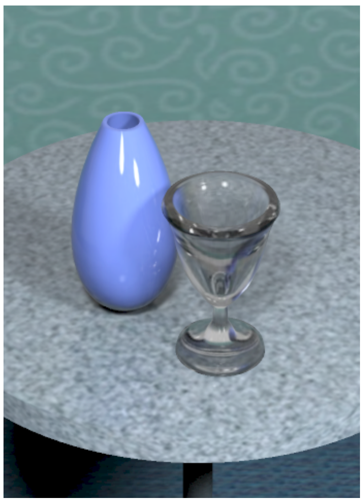
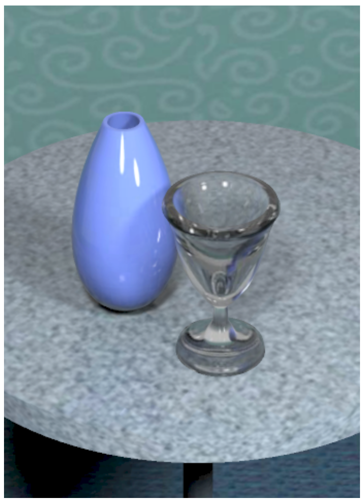
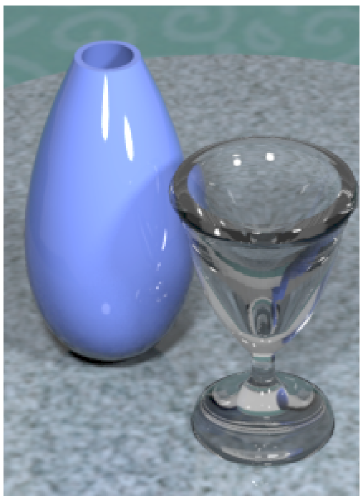
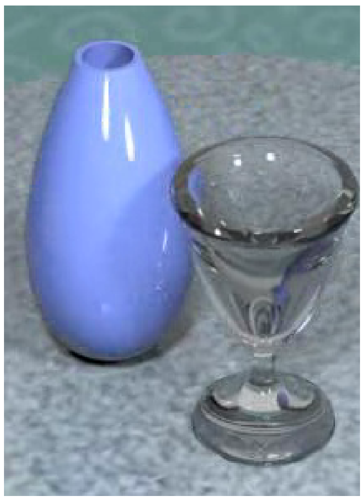
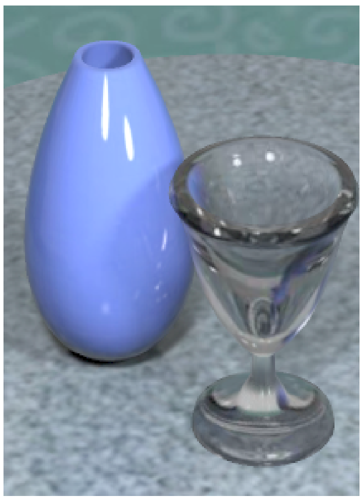
Texture-Based		
MPEG-texture 10 I-frames/block	Adaptive Laplacian SPIHT	Adaptive Laplacian DCT
		
375:1 MSE=20.3, PSNR=35.1 dB	379:1 MSE=8.66, PSNR=38.8 dB	366:1 MSE=11.8, PSNR=37.4 dB

Figure 4.9: **Demo1 Compression Results at 384:1 Target compression.** (150 KBytes/block)

	View-Based	Texture-Based
Original	MPEG-view 1 I-frame/block	MPEG-texture 1 I-frame/block
		
	355:1 MSE=13.5, PSNR=36.8 dB	384:1 MSE=10.1, PSNR=38.1 dB

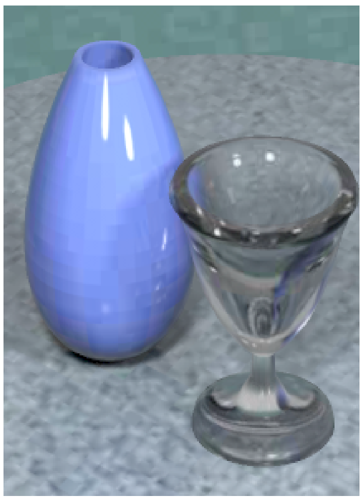
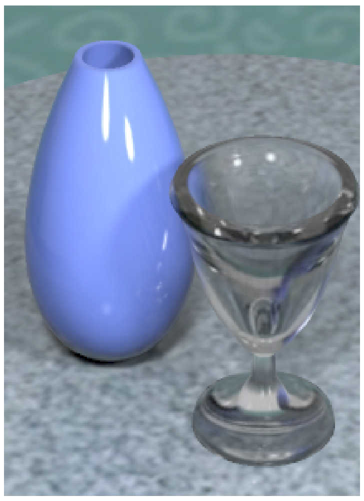
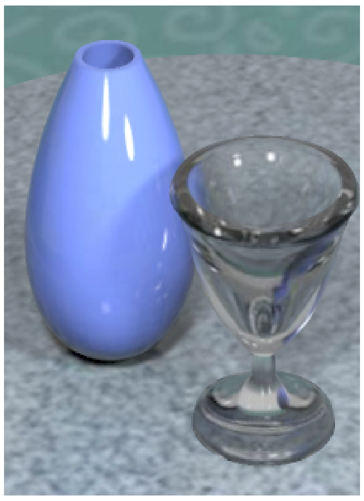
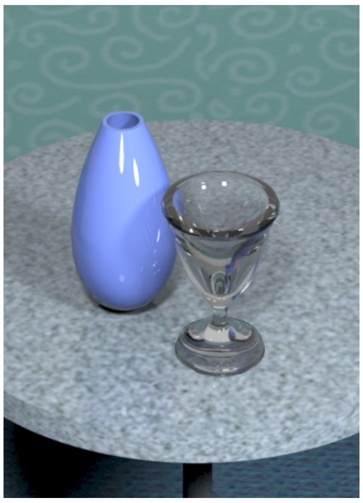
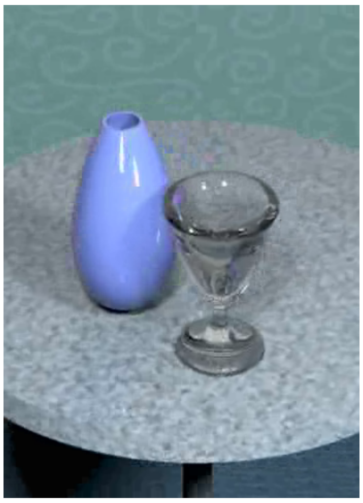
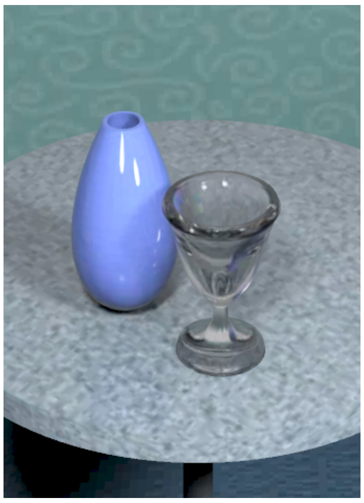
Texture-Based		
MPEG-texture 10 I-frames/block	Adaptive Laplacian SPIHT	Adaptive Laplacian DCT
		
375:1 MSE=20.3, PSNR=35.1 dB	379:1 MSE=8.66, PSNR=38.8 dB	366:1 MSE=11.8, PSNR=37.4 dB

Figure 4.10: **Close-up Demo1 Compression Results at 384:1 Target compression.** (150 KBytes/block)

	View-Based	Texture-Based
Original	MPEG-view 1 I-frame/block	MPEG-texture 1 I-frame/block
	 570:1 MSE=29.8, PSNR=33.4 dB	 760:1 MSE=25.9, PSNR=34.0 dB

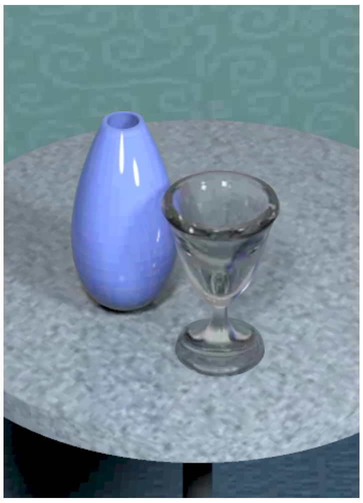
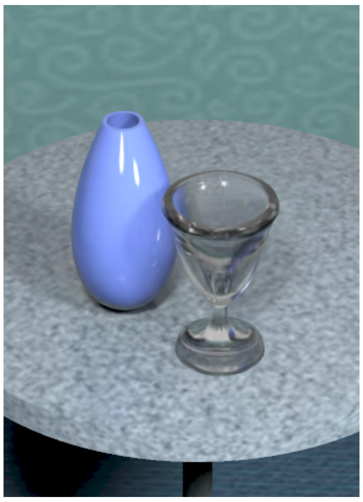
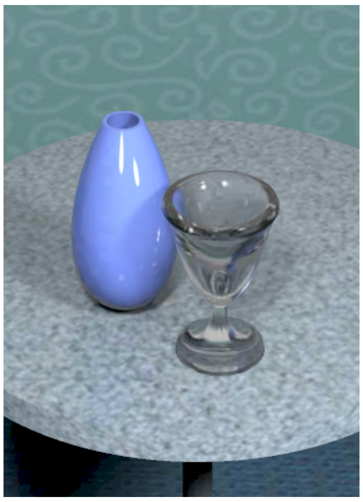
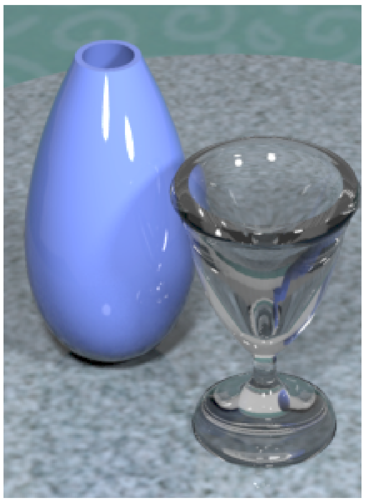
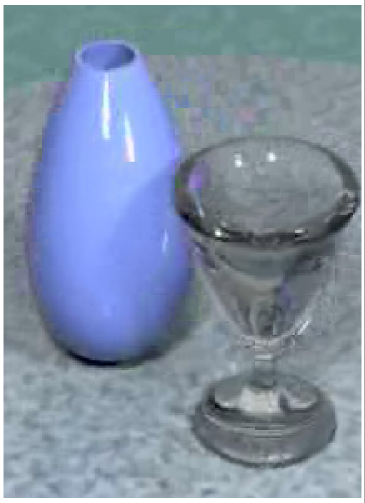
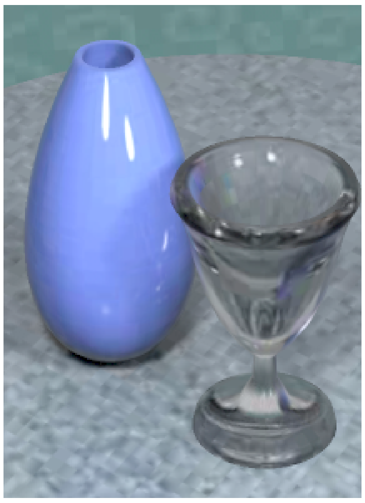
Texture-Based		
MPEG-texture 10 I-frames/block	Adaptive Laplacian SPIHT	Adaptive Laplacian DCT
 418:1 MSE=25.0, PSNR=34.1 dB	 751:1 MSE=16.5, PSNR=36.0 dB	 732:1 MSE=18.1, PSNR=35.5 dB

Figure 4.11: **Demo1 Compression Results at 768:1 Target compression.** (75 KBytes/block)

	View-Based	Texture-Based
Original	MPEG-view 1 I-frame/block	MPEG-texture 1 I-frame/block
	 570:1 MSE=29.8, PSNR=33.4 dB	 760:1 MSE=25.9, PSNR=34.0 dB

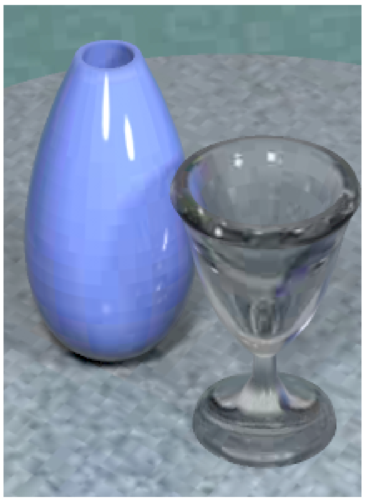
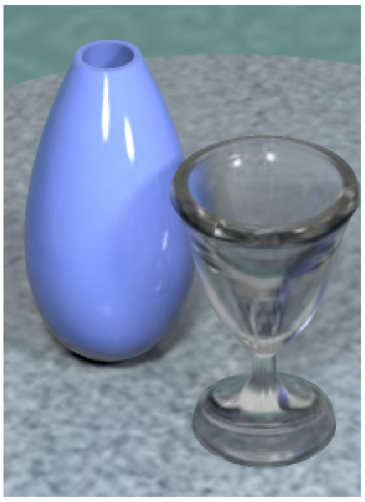
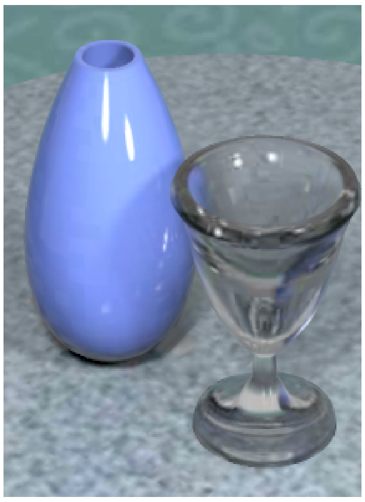
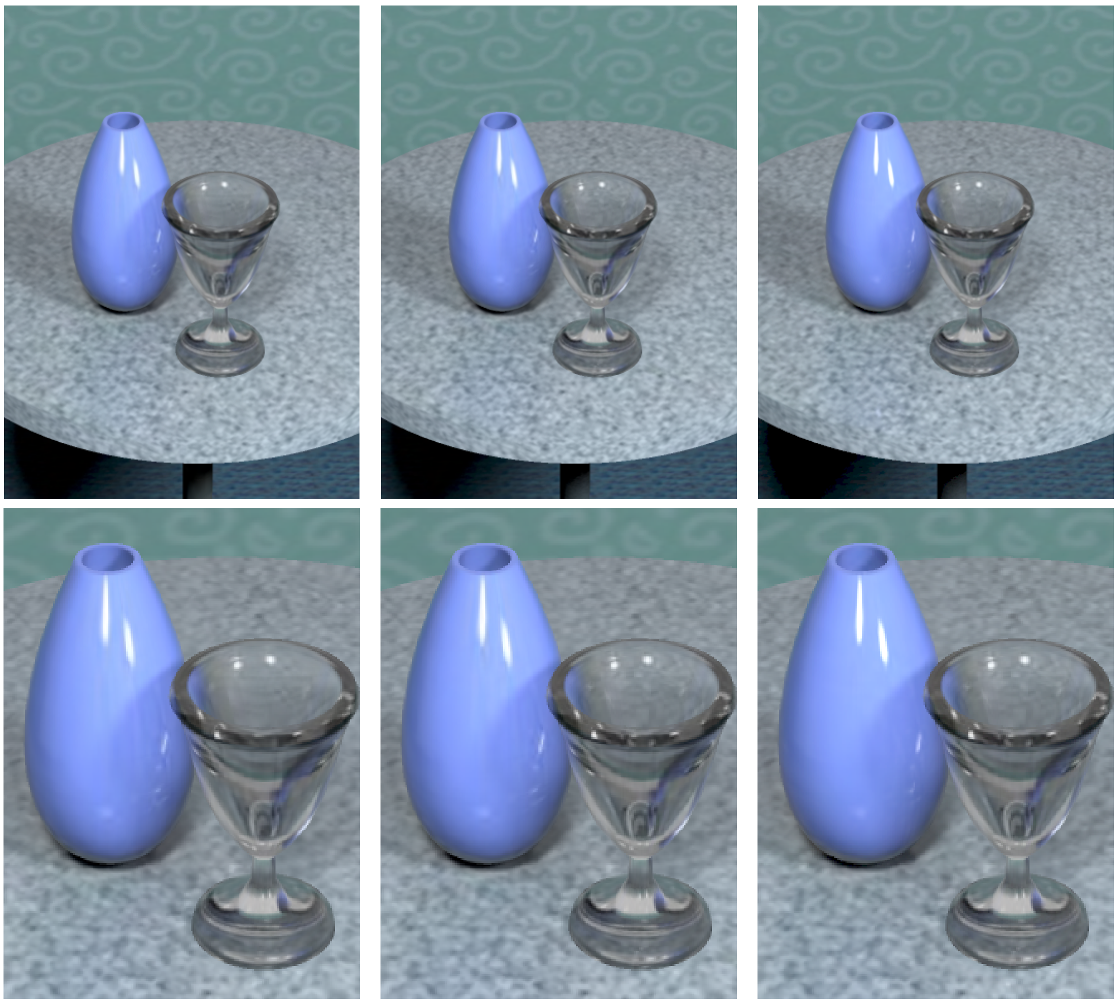
Texture-Based		
MPEG-texture 10 I-frames/block	Adaptive Laplacian SPIHT	Adaptive Laplacian DCT
 418:1 MSE=25.0, PSNR=34.1 dB	 751:1 MSE=16.5, PSNR=36.0 dB	 732:1 MSE=18.1, PSNR=35.5 dB

Figure 4.12: **Close-up Demo1 Compression Results at 768:1 Target compression.** (75 KBytes/block)

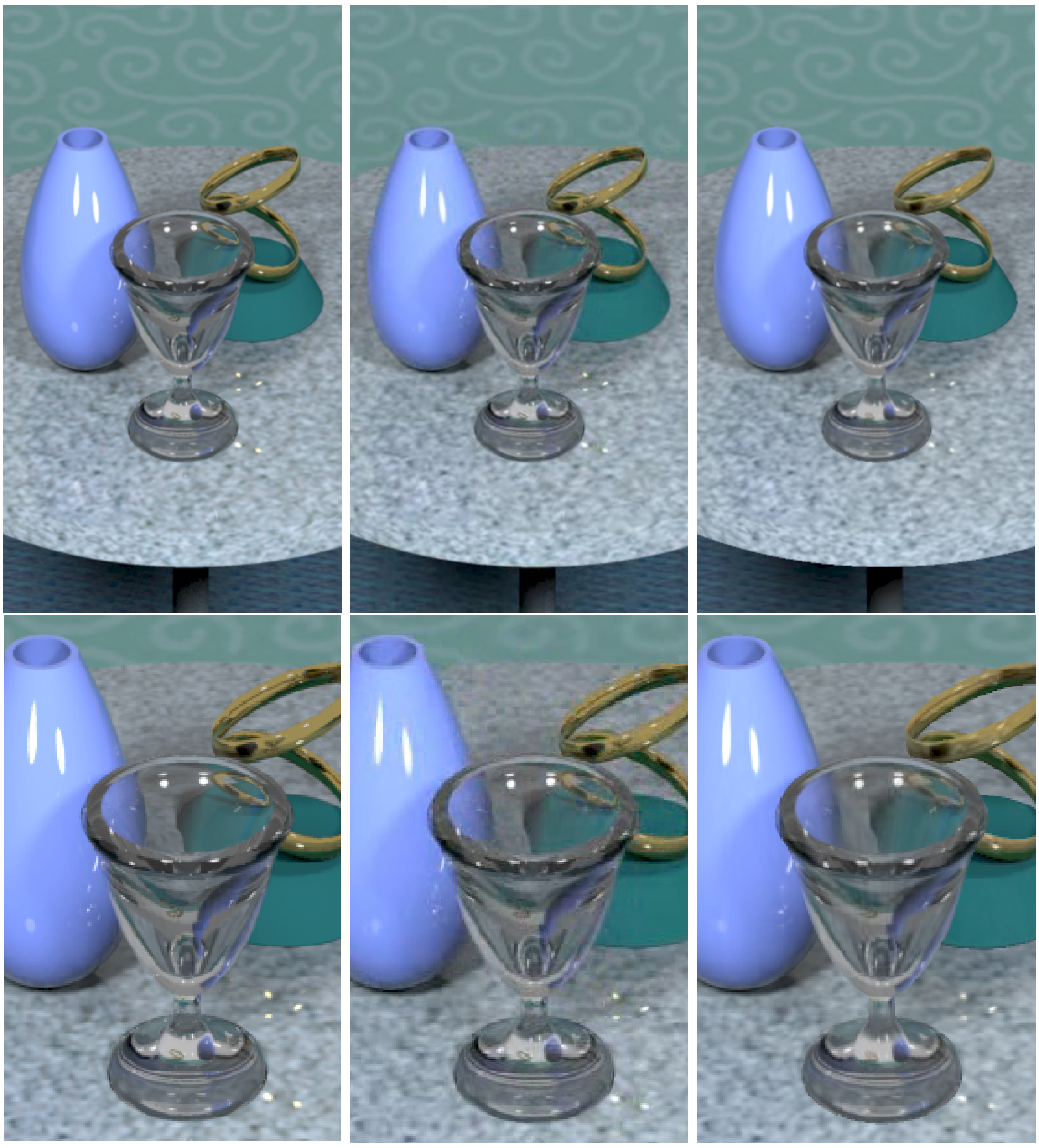


(a) Combined, non-adaptive MSE=11.5, PSNR=37.5 dB

(b) Combined, adaptive MSE=9.64, PSNR=38.3 dB

(c) Separated, adaptive MSE=10.3, PSNR=38.0 dB

Figure 4.13: **Benefits of Adaptive Subdivision and Lighting Separation.**



Original

MPEG-view
361:1
MSE=10.9, PSNR=37.7 dB

Adaptive Laplacian SPIHT
361:1
MSE=10.3, PSNR=38.0 dB

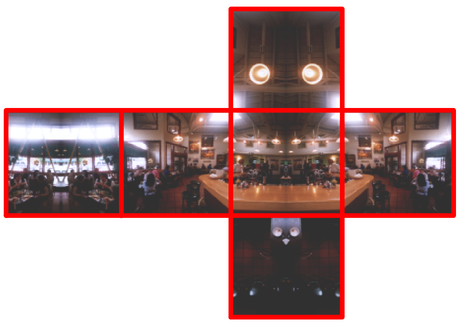
Figure 4.14: **Demo2 Compression Results at 361:1 Target compression.** (160 KBytes/block)

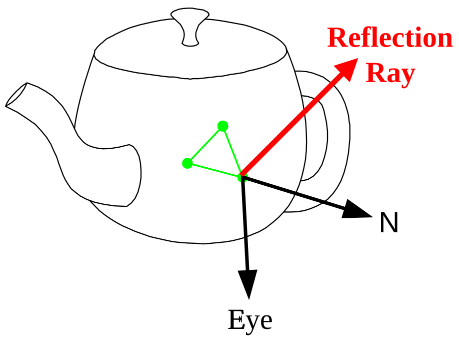
Chapter 5

Parameterized Environment Maps

In Chapter 4, we capture realistic, pre-rendered shading effects including reflections as parameterized texture maps (PTMs) on surfaces. This method of capturing view-dependent shading makes it hard to move "off the manifold" or away from the sampled views – the shading looks (and indeed is) pasted on due to the use of static texture coordinates. Traditional environment maps have been developed in graphics for handling reflective objects, but as we show in Section 5.1, such environment maps fail to capture local reflections including effects like self-reflections and parallax in the reflected imagery. We instead propose parameterized environment maps (PEMs), a set of per-view environment maps which accurately reproduce local reflections at each viewpoint as computed by an offline ray tracer. Even with a small set of viewpoint samples, PEMs support plausible movement away from and between the pre-rendered viewpoint samples while maintaining local reflections. They also make use of environment maps supported in graphics hardware to provide real-time exploration of the pre-rendered space. In addition to parameterization by viewpoint, our notion of PEM extends to general, multidimensional parameterizations of the scene, including relative motions of objects and lighting changes.

Our contributions include a technique for inferring environment maps providing a close match to ray-traced imagery (Section 5.8). We also explicitly infer and encode all MIPMAP levels of the PEMs to achieve higher accuracy. We propose layered environment maps that separate local and distant reflected geometry (Section 5.7). We explore several types of environment maps including finite spheres, ellipsoids, and boxes that better approximate the environmental geometry (Section 5.4). We demonstrate results showing faithful local reflections in an interactive viewer (Section 5.10).


(a) Static Environment Map



(b) Static EM indexing

Figure 5.1: **Traditional, or static, environment maps.** Traditional environment maps achieve a reasonable approximation of reflections and are easily supported in hardware. (a) They are constructed using standard techniques, such as by taking a photograph of a reflective sphere in an environment, or rendering six faces of cube from an object center. (b) Texture coordinates are computed for static environment maps by reflecting the ray from the eye off the local geometry and using the direction of the reflection ray to index into the map, like the one shown on the left. This is in contrast to texture maps where the texture coordinates are fixed. Note that we use the abbreviation EM for environment maps.

5.1 Traditional/Static Environment Maps

Accurate, real-time rendering of shiny objects has long been a goal of computer graphics. Environment maps (EMs) [Bli76, Gre86, Hae93, Mil84, Voo94], which store a sphere of radiance incident at a point, achieve a reasonable approximation of reflections and are easily supported in hardware. Unfortunately, because EMs, illustrated in Figure 5.1, are constructed from a single point like the reflective object's center, they fail to accurately reproduce *local* reflections (Figure 5.2c). Self-reflections are lost since the object itself is omitted during EM construction and geometric accuracy suffers when the surface point is not exactly at the object center or if the reflected object is not very distant.

5.2 Parameterized Environment Maps

Our solution to the problem of accurately reproducing local reflections is to record multiple EMs at a set of viewpoints in a pre-rendered space. These EMs are not spherical images of the environment at a point. Instead, they are inferred as a least-squares best match to a ray-traced image taken at each viewpoint, *when applied as an EM in a rendering by the target graphics system.* The result is

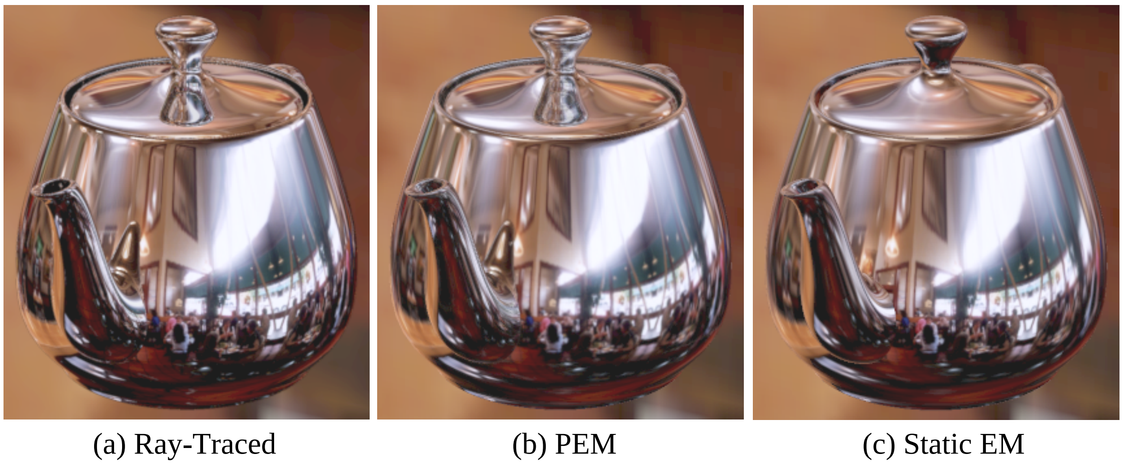


Figure 5.2: **Simulating Reflections.** Note the missing self-reflections of the knob and spout in (c) and the fidelity of the PEM (b) matched to the ray-traced image (a).

a sequence of EMs, which we call a parameterized environment map, or PEM (Figure 5.3). More generally, PEMs can be parameterized with any number of dimensions, which can control positions of objects and lights as well as view, and any number of samples per dimension.

Using PEMs, we are able to closely match local reflection effects like self-reflections evident in each ray-traced sample (compare Figure 5.2a and 5.2b). Furthermore, we can move the viewpoint away from the ray-traced samples, plausibly maintaining these local reflection effects. In fact, by inferring PEMs from ray traced images along only a 1D subspace of views, we achieve convincing local reflections from an entire 3D viewspace quite far from the original samples. Capturing a 2D viewspace provides even better accuracy but uses many more maps. PEMs also capture much of the coherence in view-dependent shading, and thus compress very well.

With simple EM models, this approach does not necessarily match the ray traced images exactly, since the mapping from the surface of the reflector to the EM is not necessarily one-to-one. In other words, two or more reflected positions in the world can map to the same EM point. [1] We reduce these conflicts by using *layered* EMs which separate local and distant parts of the environment. The final result is superior to images generated using a static EM.

[1]Another difficulty is that the same point in the world can be multiply imaged by the reflector with different shading if its surface is non-Lambertian.

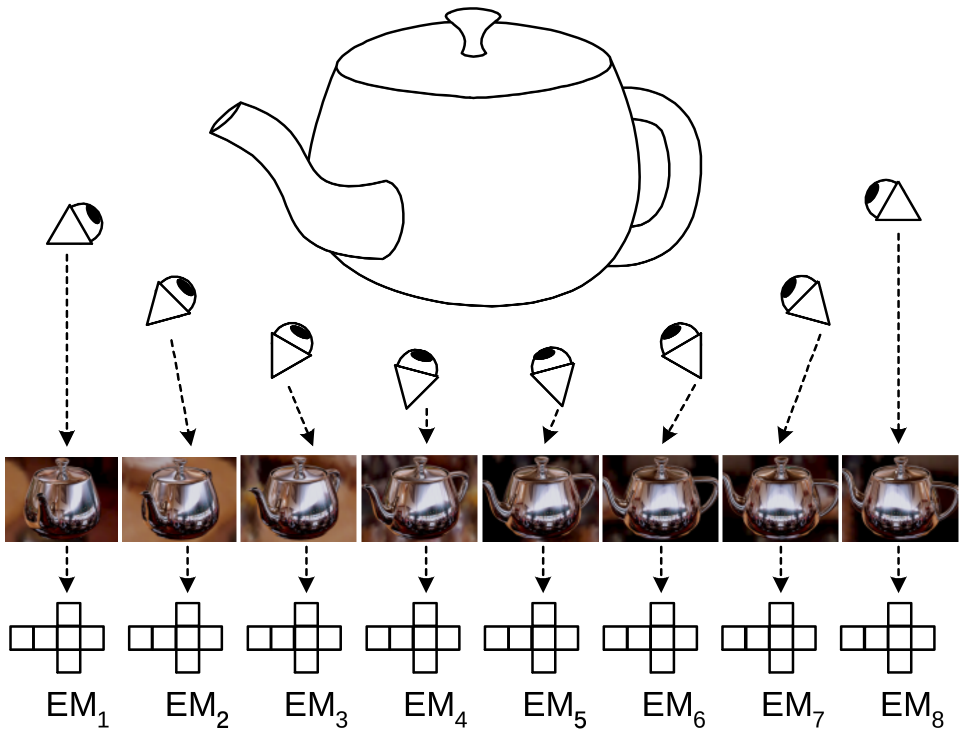


Figure 5.3: **Parameterized Environment Map.** A PEM is a sequence of environment maps (EMs) recorded over a set of viewpoints (or other parameters). These environment maps are computed by inverse fitting to ray-traced images of the reflective object pre-rendered at sample locations in the parameter space. The diagram suggests a cube map parameterization for each EM, but other choices are possible.

5.3 System Overview

We begin with a ray-traced image at each viewpoint as in Figure 5.3, or more generally, each point in the parameter space which is to be interactively explored later. The ray tracer segments the imagery by separating the images of individual objects (Figure 5.8). It also segments various shading terms on each object, including a diffuse layer, Fresnel reflection modulation layer, and an incident specular layer. We use a modified version of Eon, a Monte Carlo distribution ray tracer [Coo84a, Shi92, Shi96].

Given the parameterized, segmented image layers for each object, we compute, or *infer*, parameterized texture and environment maps that accurately reproduce the ray-traced images when applied to the original geometry by graphics hardware. PTMs are inferred for the diffuse layer using the technique described in Chapter 3. PEMs are inferred for the incident specular layer and are handled by applying the that technique to hardware rendering with environment maps, as discussed

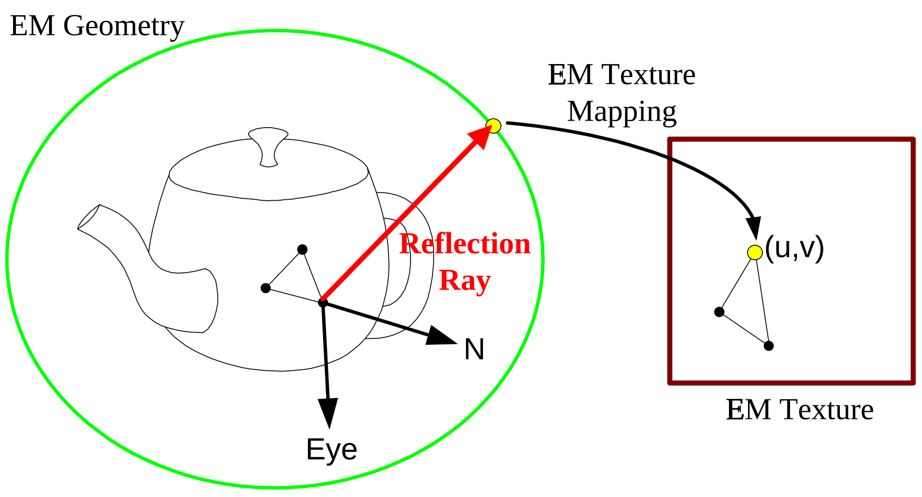


Figure 5.4: **Environment Map Geometry.** We show how environment map texture coordinates are generated for an individual vertex. This is done by intersecting the reflection ray with the simple geometry that approximates the environment, in this case an ellipsoid. We then map this intersection point using the environment texture mapping to find its (u, v) texture coordinates.

in detail in the next section. Because of the segmentation, the PTMs and PEMs exhibit much coherence and can be greatly compressed, using schemes like MPEG for 1D parameter spaces, or the multidimensional Laplacian pyramid described in Chapter 4.

At run-time, as a user explores the space, the system decodes a per-object texture and environment map sample closest to the user's current parameter location (e.g., viewpoint). The resulting maps are then loaded into the graphics hardware. Using appropriate texture blending modes, we recombine the diffuse layer, if present, with the product of the Fresnel-modulation layer and environment-mapped result. This produces a rendering that accurately matches the ray traced imagery at the sampled parameter locations, and successfully interpolates images away from those samples. We also blend between neighboring parameter samples rather than choosing the closest to provide a smoother result.

5.4 Environment Map Representations

It is important to distinguish the geometry of an EM from its parameterization (Figure 5.4). An EM's *geometry* refers to how it approximates the reflecting environment. For example, the environment can be approximated by a sphere at infinity, by a finite cube, or by a finite hemisphere with planar bottom. By picking an EM geometry that closely matches the actual environment's, we obtain better predictions of how reflections move as the view changes. Of course, as we move to even better approximations of the environment, such as light fields, LDIs, or even the precise geometry indexed by ray-tracing, the cost of computing samples becomes impractically high.

The *parameterization* of an EM refers to how the geometry is represented in a 2D map. For example, the infinite sphere can be represented using a latitude/longitude parameterization, the gazing ball (or OpenGL) parameterization, or by six faces of a cube. The choice of EM parameterization is less crucial than the choice of geometry in determining final accuracy, but does impact how much resolution the maps will require. Note that confusion arises because parameterizations tend to have names related to shape; for example, a finite sphere geometry can be parameterized using a cube parameterization and vice versa.

We have tried several types of EM geometry achieving best results with finite rectangular parallelepipeds, called *box maps*, and finite spheres and ellipsoids. Box maps are useful for room environments. For objects resting on flat surfaces, it is effective to align the box bottom with this flat surface and extend the other sides to match the average distance to environmental geometry. Finite ellipsoid maps have proved useful for local reflected geometry (see Section 5.7).

To index such maps, we resort to a combination of software and hardware texture coordinate generation. Graphics systems, such as the Nvidia GeForce running under DirectX, currently support only the infinite sphere geometry with cube or gazing ball parameterization. In software, for each polygonal mesh vertex, we bounce the view ray off the vertex using its associated normal to determine a reflecting ray. The resulting ray is intersected with the EM geometry, such as a finite box, sphere, or ellipsoid. The simplicity of these models makes the ray intersection calculation practical for real-time applications. Taking this point of intersection and subtracting the object's EM origin point yields a vector that is used as the hardware EM index (normalization isn't required for the cube parameterization). The EM origin is chosen as an area-weighted average of the object's triangle centroids.

The hardware-supported cube map parameterization can thus be used for any of these simple geometries. [2] Alternatively, direct use of the 6 cube face texture maps by treating each face as an independent texture would require expensive texture state switching and explicit handling of triangles whose vertex indices straddle the cube edges or corners. The cube map parameterization is useful since the hardware treats the 6 cube faces like a single texture, the hardware automatically handles triangles that straddle cube edges or corners, and the parameterization does not have a singularity. The singularity in the gazing ball parameterization makes it inappropriate for generating views off the ray-traced samples, as observed in [Hei99b].

[2]Note that the cube map parameterization (6 equal-sized square faces) uses texture area inefficiently for ellipsoids, but is a limitation of current hardware.

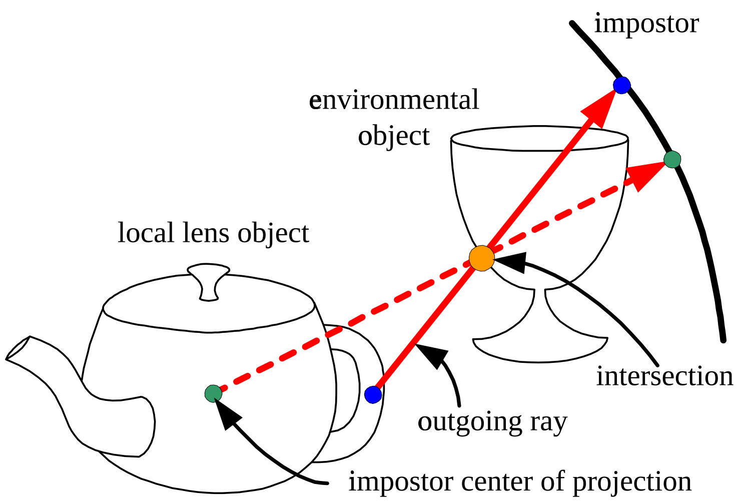


Figure 5.5: **Why Parameterize Environment Maps?** See discussion below for explanation.

5.5 Why Parameterize Environment Maps?

Our approach is to fit an environmental impostor so as to match a ray traced image at a particular view. The alternative is to use some a priori projection method such as projecting the environment onto a sphere centered at some point in the reflective object. There are two benefits to our fitting approach. It can compensate for the geometric error caused by approximating the environment with simpler geometry that acts as an environmental impostor. In the diagram in Figure 5.5, an outgoing ray, drawn in solid red, exits the teapot and strikes a cup object in the environment. Using fitted impostors, the color of this cup is correctly stored at the blue point on the impostor, so it can be accessed when the same outgoing ray is generated at run-time. If we instead simply projected the cup onto the sphere, then the orange intersection point would be incorrectly mapped to the green point on the impostor. This is the point where the line between this intersection and the impostor center of projection hits the impostor.

A second benefit is that our approach captures view-dependent shading in the environment. The outgoing ray that intersects the cup object is generated from the desired viewpoint. So if the cup object is itself reflective or refractive then the color value stored will result from the incoming ray with the correct direction. Spherical projection just uses rays emanating from the center of projection, which are view-independent and thus dont account for view-dependent shading on the cup.

By inferring a separate environment map at each viewpoint, we capture view-dependent shading in the environment and compensate for geometric error, and thus obtain better fidelity at the ray-traced samples.

5.6 Comparison with Alternative Approaches

Accurate reflections between arbitrary objects can be produced by ray-tracing [Whi80], but at significantly higher cost than traditional texture-mapped polygon rendering. This has led to efforts that exploit fast graphics hardware to produce realistic reflections. Reflections on planar surfaces, explored by Diefenbach [Dief96], can be achieved using a rendering that mirrors the viewpoint about the reflection plane. Reflections on curved objects, studied by Ofek and Rappoport [Ofek98], can be performed by transforming each vertex in the reflected image with respect to the reflector's geometry. This scheme handles smooth reflecting objects that are either concave or convex; objects with mixed convexity or saddle regions require careful decomposition.

Image-based rendering (IBR) methods such as the Light Field [Lev96] and Lumigraph [Gor96] reduce the plenoptic function to a tabulated 4D field. These view-based IBR methods perform linear interpolation for each ray in the plane of camera viewpoints when rendering novel views not captured in the source images, and thus do not model how reflections move as the view changes between sample positions. Other view-based methods that projectively map source images onto approximate geometry, such as view-dependent textures [Deb98a] and unstructured lumigraph rendering [Bue01], also linearly interpolate between camera viewpoints either by distance between the desired ray and camera positions or by angle between the desired ray and rays emanating from camera centers that converge to the same point on the approximate geometry. These view-based methods likewise ignore parallax in the reflected imagery. The net result is that a denser sampling of views is required than with our layered PEMs approach to avoid blurriness and ghosting artifacts.

Surface light fields [Mil98a, Woo00] parameterize the radiance field over surfaces rather than views to better capture spatial coherence, especially when surfaces are mostly diffuse. For mirror-like reflectors, the surface light field is essentially identical to a sphere-at-infinity EM per surface point. We obtain a better prediction of how reflections change with view by using more accurate geometric approximations of the environment (Section 5.4) including simple finite ellipsoids and boxes, and by separating reflections into multiple layered maps for local and distant elements. We also associate a single EM per object, but parameterized by view, rather than multiple, view-independent ones over a dense set of its surface points.

When applying existing IBR methods to highly reflective surfaces, it is not clear whether the required sampling density of views (for view-based IBR) or of emitted radiance per surface point (for surface-based IBR) is practical. Current results demonstrate only fairly blurry highlights from light sources. In Wood et al. [Woo00] for example, this is not surprising given that the 258 lumisphere samples used per point represent 2-3 orders of magnitude fewer samples than typical EMs contain. Furthermore, to reconstruct an image from a particular view requires visiting an irregular scattering of samples over the entire 4D light field. With PEMs, all the information needed to reconstruct a particular view is spatially coherent in the form of a single EM image (two are needed for smoother interpolation in a 1D viewspace) whose access is already supported by hardware. PEMs also handle arbitrary view subspaces (e.g., 1D ones) and other scene parameterizations.

Heidrich et al. [Hei99a] decouple geometry from illumination by using a light field to map incoming view rays into outgoing reflected or refracted rays, thus handling self-reflections. These outgoing rays then index either a static environment map, which ignores local effects further from the reflector, or another light field representing the environment, which is more accurate but also more costly. The result allows independent change to the reflecting object geometry and the environmental radiance, but suffers from the limitations of other IBR methods mentioned above.

Cabral et al. [Cab99] also decouple the reflecting object from the illumination. They store a collection of view-dependent EMs where each EM pre-integrates a specific BRDF with a lighting environment. The lighting environments for these EMs are generated using standard techniques, such as taking photographs of a physical sphere in a desired environment or rendering the six faces of a cube from the reflecting object center using a ray-tracer. As a result these EMs suffer from the same problems as traditional EMs in handling local reflections.

Bastos et al. [Bas99] reproject LDIs into a reflected view for rendering primarily planar glossy surfaces in architectural walkthroughs. Lischinski and Rappoport [Lis98] propose two ideas: layered light fields, which are a collection of view-dependent LDIs for glossy objects with fuzzy reflections, and image-based ray-tracing for sharp reflections, where rays are traced through three view-independent LDIs that accurately represent the scene geometry. Our approach succeeds with much simpler and hardware-supported EMs rather than reprojecting LDIs or ray-tracing through LDIs.

As mentioned in the introduction to this chapter, our technique of Chapter 4 captures realistic, pre-rendered shading effects including reflections as parameterized texture maps (PTMs) on surfaces. This method of capturing view-dependent shading makes it hard to move “off the manifold” or away from the sampled views – the shading looks (and indeed is) pasted. In this chapter, we apply the same method of texture inference, but instead of texture maps we compute EMs in which a reflection ray is actually bounced off the surface and intersected with a simple approximation of the environment. This results in much better quality off the manifold.

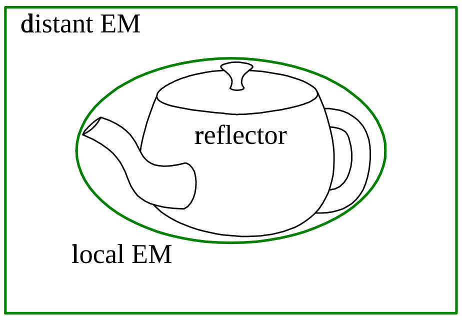


Figure 5.6: **Layered Environment Map.** Here a box geometry is used for the distant EM, modeling more distant reflected parts of the environment, while an ellipsoidal geometry is used for the local EM, modeling parts of the reflector itself present in self-reflections.

5.7 Layered Environment Maps

Segmenting the environment into separate maps for local and distant elements better approximates how each elements' reflections behave (Figure 5.6). The separation allows different EM geometries to be used to approximate the environmental geometry in each layer. It also supports parallax between imagery in each layer.

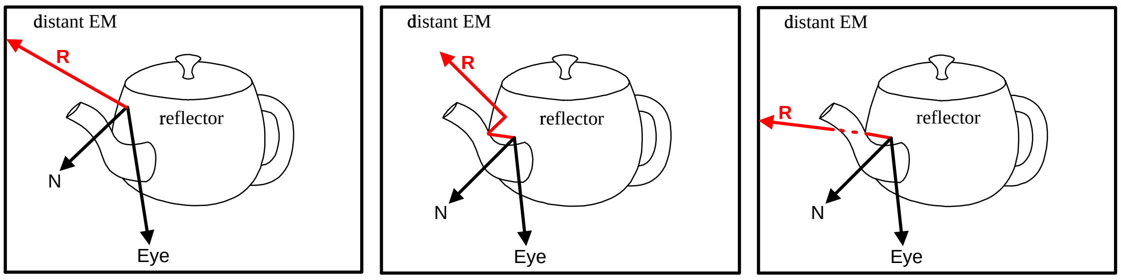
To perform this separation, the ray tracer records separate layers for rays which bounce off a reflective object and immediately reach the distant environment and rays which bounce one or more additional times off the object (Figures 5.7 and 5.8). EMs are then inferred for each layer separately. We have found that a finite ellipsoidal EM geometry works well for the local layer. An ellipsoid is selected to tightly bound the reflecting object. We use an axis-aligned bounding ellipsoid centered at the object centroid with axis scales that minimize resulting volume, determined using brute force optimization.

To handle occlusion effects, the local EM is computed as a 4 channel image with transparency representing fraction of coverage in the local layer from an antialiased rendering. To compute the distant layer without local occlusions, the ray tracer propagates rays through the reflecting object if they intersect after the first bounce, thus defining all distant layer samples without need for an alpha channel. At run-time, we use the over blending mode to composite the local layer (subscript L) over the distant (subscript D) before modulating by the Fresnel term, F, via

$$(\alpha_L rgb_L + (1 - \alpha_L) rgb_D) F$$

Note that this method easily generalizes to more than two environmental shells.

Because the local EM geometry only approximates the local geometry being reflected, erroneous



(a) Ray directly reaches distant environment

(b) Ray bounces more than once off reflector

(c) Ray propagated through reflector

Figure 5.7: **Constructing the Distant Environment Map Layer.** The distant layer is constructed in the ray-tracer from rays that immediately reach the distant environment (a). However, it is possible for rays that bounce off the object, to hit the object again after their first bounce (b). For the distant layer, we ignore secondary bounces off the reflective object by allowing rays to pass through the object after their first bounce (c).

local reflections can be generated. We minimize such problems by separating polygons on the reflector that lie on its convex hull from those lying inside it as a view-independent pre-process. At run-time, polygons lying on the convex hull are rendered only with the distant EM, since they can never exhibit self-reflections. The rest of the polygons are textured using the local/distant EM combination.

The incident specular layers for a shiny object represent the incident light that is then attenuated by the surface reflectance and reflected towards the viewer. Separating out the view-dependent Fresnel modulation by recording incident rather than emitted radiance makes the layers more view independent and thus more coherent. After generation by the ray tracer, the layers are captured as EMs. Our factorization of the Fresnel modulation layer from the incident specular radiance and use of graphics hardware for its evaluation is based on the work of Heidrich and Seidel [Hei99b]. The next subsections present our approach for inferring EMs. By running the inference method at each point in parameter space, we obtain PEMs which can then be compressed.

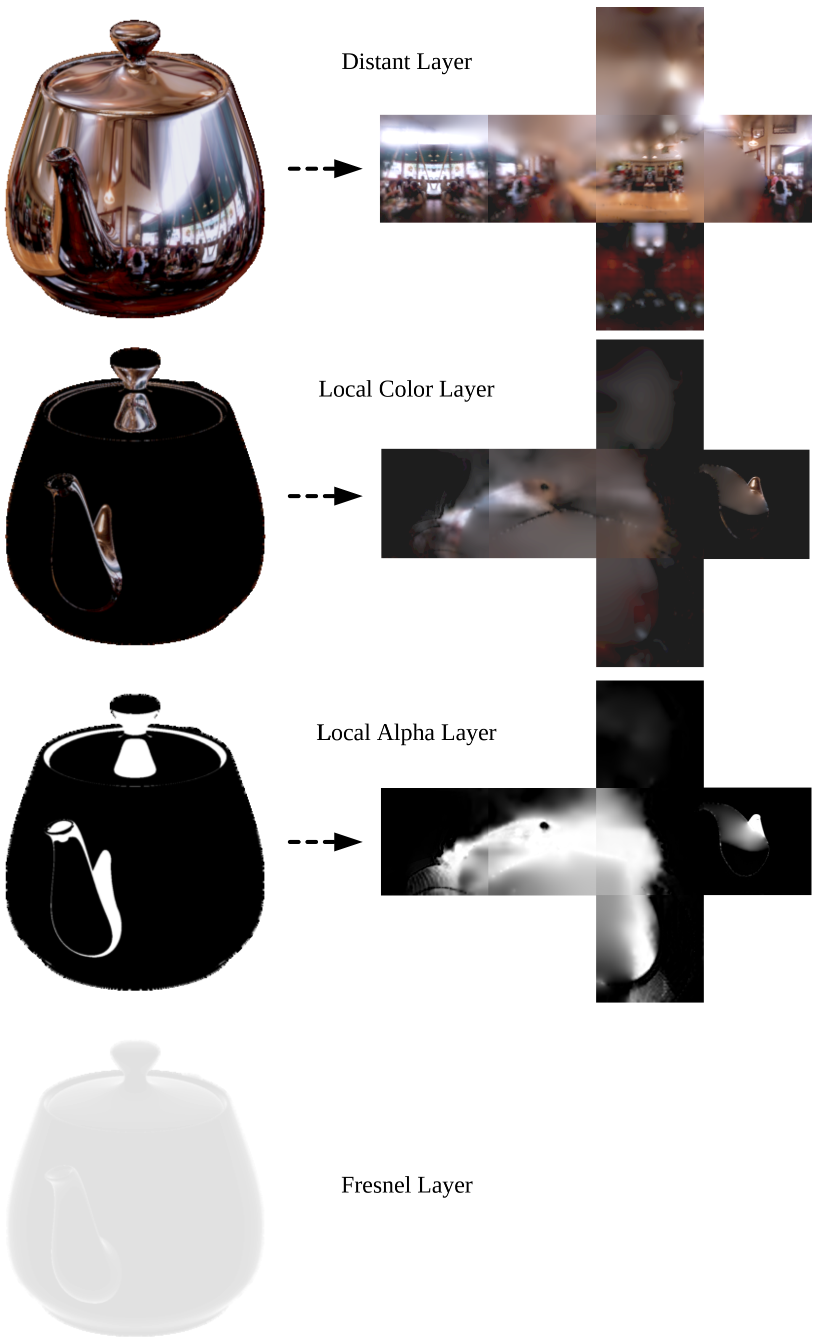


Figure 5.8: **Segmented Layers Produced by the Ray Tracer for one Viewpoint and Corresponding Inferred EMs.** The environment maps have a cube map parameterization (6 faces) and consist of MIPMAPs for each face (not shown). The ray-tracer keeps the highly view-dependent fresnel modulation separate from the incoming radiance. We generate the fresnel modulation at run-time using a simple formula.

5.8 Inferring Layered Environment Map Texture Maps

We use the least-squares inference approach described in Chapter 3. The only difference in computing an EM instead of a texture is that the rendering matrix is created using test renderings with environment mapping instead of texture mapping.

5.8.1 Environment Map MIPMAPs

Our inference method solves for all MIPMAP levels of the EM simultaneously. Interestingly, we find that these levels are not simple filtered versions of each other, even when performing weighted filtering that accounts for the variation of solid angle over the EM parameterization, as in [Hei99b]. This is because in computing the best matching EM at a particular viewpoint, our inference method implicitly accounts for the reflector geometry, which can magnify and distort the reflection in a spatially varying way. Therefore, unlike in Chapter 4, we explicitly encode all levels of the EM MIPMAP rather than creating them on-the-fly as decimated versions of the finest level. The result is improved sharpness in the reflections. To increase compression, encoded levels are stored as residuals from corresponding decimated versions of the finest level.

5.8.2 Environment Map Resolution

Choosing the proper EM resolution is important to preserve frequency content in the reflections. A very conservative approach is to use test renderings to determine the most detailed EM MIPMAP level actually accessed by the graphics system. Texture memory bandwidth and capacity limitations may dictate the use of somewhat lower resolutions.

5.9 Runtime System

The Fresnel modulation layer is generated on-the-fly using a per-vertex software shader that copies the ray tracer's computation of the Schlick model [Sch93]. We make use of a 1D texture map to better interpolate the fifth order polynomial involved in that model. Such per-vertex computation requires adequate tessellation of the reflecting object's geometry.

We use multi-pass rendering to assemble the shading layers. Purely reflective objects in a 1D viewspace require 5 texture map accesses: 2 EMs for the local/distant dual at each of 2 viewpoints for smooth interpolations and the 1D Fresnel map. Addition of a diffuse layer requires one more texture access. Current PC graphics hardware, such as the Nvidia GeForce 256 and GeForce2 graphics accelerators, performs 2 texture accesses per pass, so 3 passes are needed.

Factoring surface reflectance from incident radiance is problematic on current hardware with fixed point 8-bit texture arithmetic; it is difficult to fit the dynamic range needed by the incident specular layer [Deb98b]. We clip samples that are too bright, sometimes resulting in artificially dimmed highlights. Solving this problem will require more dynamic range in texture processing, perhaps using a floating point representation.

5.10 Results

We tested our approach on a simple scene of a reflective teapot in a room environment. The teapot contains ˜40k triangles and was ray traced from a 1D viewspace partially circling the teapot at 1° per view sample to generate a PEM containing 100 EMs. We used the local/distant dual EM with finite ellipsoid for the local map and box map for the distant at $256 \times 256 \times 6$ resolution. To improve accuracy, we decomposed the teapot into two parts: the lid in one and the spout, body, and handle in the other. Each part used a separate layered EM model. A more efficient approach would be to decompose the teapot only for the local EM layer while using a single distant EM for the combined teapot.

Our viewer performs at about 17.5 frames per second with blending between adjacent viewpoints on 733MHz PC with Nvidia GeForce 256 graphics accelerator. Downloading textures into hardware memory is a performance bottleneck; this will be alleviated by higher bandwidth memory and hardware-supported decoding of compressed textures. The following table details run-time performance, assuming the EM textures are already resident in system (but not video) memory. “Texgen time” below is the time to compute EM coordinates using ray intersection with simple box and ellipsoid primitives, performed on the CPU. Use of programmable vertex shaders supported in the graphics system, such as those available in DirectX DX8, may speed up this computation.

	On the Manifold (unblended)	**Off the Manifold (blended)**
#geometry passes	2	3
texgen time	35ms	35ms
frame time	45ms	57ms
FPS	22	17.5

Table 5.1: Parameterized Environment Map Runtime Performance

For each viewpoint sample, ray tracing required about 15 minutes while EM inference took 5.5 minutes.

Figure 5.2 compares results “on the manifold”; i.e., at the ray traced samples. Our PEMs achieve good fidelity to the ray traced images matched, while static EMs eliminate local effects.

To test off the manifold quality, we tried several alternatives, including PTMs and non-layered PEMs using a single sphere-at-infinity EM geometry. Figure 5.9 compares these choices for EM geometry at a viewpoint between the original samples. A ray tracing at the exact viewpoint is included for comparison. It can be seen that the layered EM model produces greater accuracy than the sphere-at-infinity EM (notice especially the reflection of the knob on the lid). The results are even more obvious interactively than in a static image, where the dual layered PEM model eliminates “wobble” exhibited by the simpler model as the user moves off the manifold.

Figure 5.10 compares results between PEMs and PTMs above the plane of viewpoint samples. Note especially the lack of fidelity of the PTM image in the teapot lid reflections. These images and, more dramatically, the interactive results, show the pasted-on effect obtained by PTMs off the manifold, resulting in a popping artifact when switching between textures inferred at adjacent viewpoint samples. PTMs also suffer from disocclusions where non-contributing texture area is revealed in a different view. While disocclusions can also occur with PEMs since they too are solved at a given view, they are less frequent and less visible as long as the object reflects most of its environment.

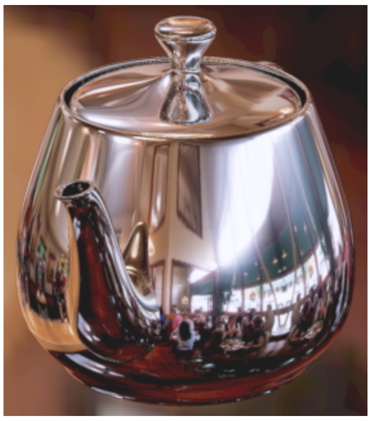
Thus far, we have presented results for a mirror-like specular teapot. Glossy objects can be handled in our approach simply by inferring environment maps in the same way, but with respect to ray-tracing with glossy materials. At run-time, we compute the reflection ray at each vertex as the principle direction of the glossy reflection, and find its intersection with the simple geometry that approximates the environment to compute texture coordinates. Figures 5.11 and 5.12 compare results between ray-tracing, layered PEMs and single sphere-at-infinity PEMs for a glossy teapot. Layered PEMs achieve good fidelity to the ray-traced images at the sample viewpoints and plausible results between and away from the sample viewpoints. This is in contrast to the sphere-at-infinity PEM results which are noticeably less accurate at the sample viewpoints (notice the reflection of the knob on the lid), and especially away from the sample viewpoints, where the reflection of the knob on the lid is absent, and the reflection of the spout on the teapot body is exaggerated. Like the mirror-like teapot, the results are even more obvious interactively than in a static image, where the dual layered PEM model eliminates “wobble” exhibited by the simpler model as the user moves off the manifold.

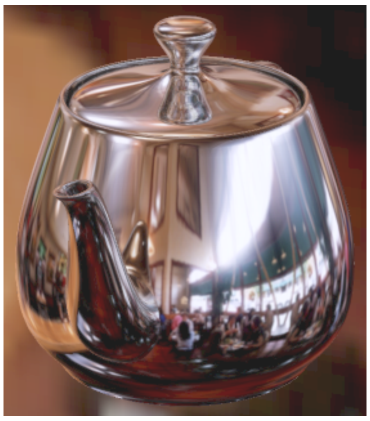
5.11 Discussion

PEMs provide a faithful approximation to ray-traced images at pre-rendered viewpoint samples and the ability to plausibly move away from those samples using real-time graphics hardware.

To achieve our results, we decomposed the teapot into two parts, the lid in one and the spout, body, and handle in the other, and used separate layered environment maps for each part. In particular, separate ellipsoids were tightly fitted to the geometry for each part to act as impostors for the local layers. Decomposing an object has two benefits. First, by using separate EM impostors, we can better approximate the actual geometry in each part. Second, we reduce the likelihood of conflicts that can occur when two or more reflected positions in the world map to the same EM point. By decomposing the object, and using separate EMs for each part, we eliminate EM conflicts between the reflected images seen in each part. Rather than simply decomposing the teapot into two parts, one might consider decomposing the teapot further into smaller pieces to achieve higher fidelity to the ray-traced images. If we take this idea to its logical extreme, then we essentially end up with a surface light field approach where each point on the reflective teapot has its own environment map, or lumisphere. It is interesting to note that our results are quite good with a decomposition of the teapot into just two parts, obviating the need for further decomposition. However, for an arbitrary reflective object it may be necessary to perform a higher level of decomposition to achieve a faithful approximation to ray-traced images. One area for future work is automatic methods of finding the minimal amount of decomposition necessary to achieve perceptually acceptable results.

Another area which we have not addressed is handling cases where the reflector does not image parts of its environment that can nevertheless be seen in nearby views “off the manifold”. For example, a reflector can be partially occluded or occlude itself (like the teapot spout obscuring its body), thus potentially eliminating from its EM the part of the environment reflected in this occluded portion. This problem can be solved by inferring EMs for all front-facing parts of the reflector, even if they are occluded in the particular view sample. Incomplete imaging of the environment also occurs when the reflector’s set of normals incompletely cover the sphere such as with non-closed objects (like a small portion of a sphere) or objects with zero curvature (like a plane or cylinder). We address the general problem of filling in the missing portions of the environment in the next chapter on hybrid rendering.


Ray-Traced


Layered PEM

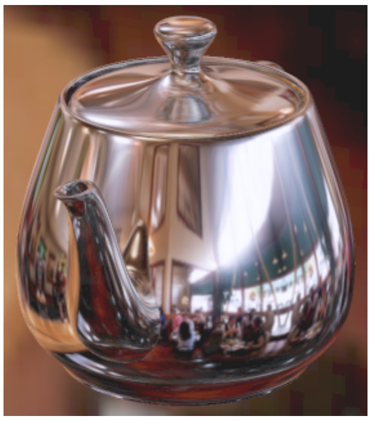
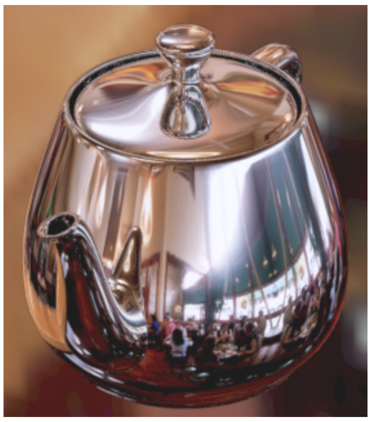
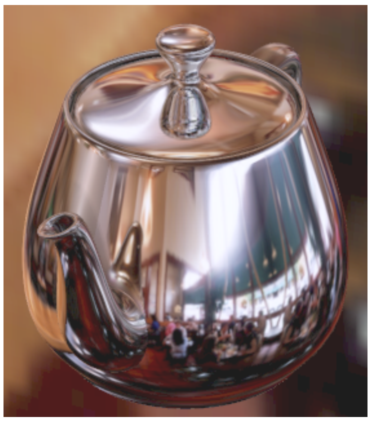
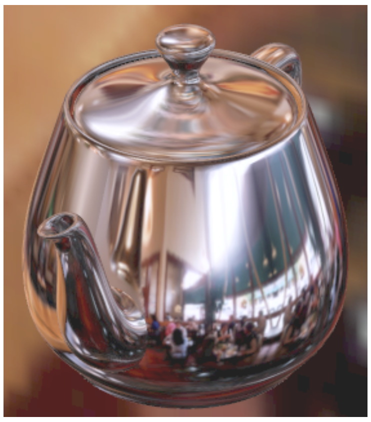

Single Sphere-at-Infinity PEM

Figure 5.9: **Results Between Viewpoint Samples.**


Ray-Traced


Layered PEM


Single Sphere-at-Infinity PEM

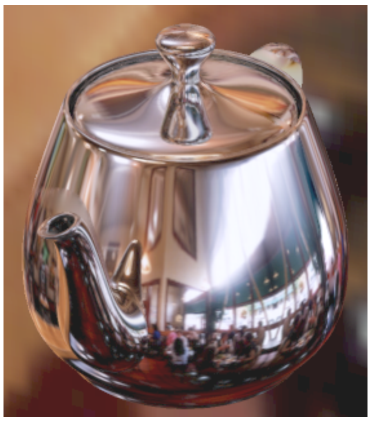

Parameterized Texture Maps

Figure 5.10: **Results Above Viewpoint Samples.**

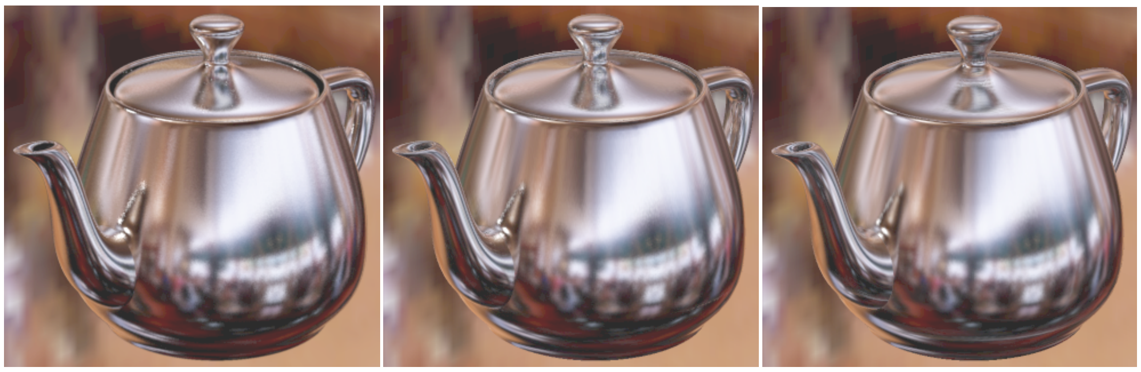


Figure 5.11: **Results At Viewpoint Samples for a Glossy Teapot.**

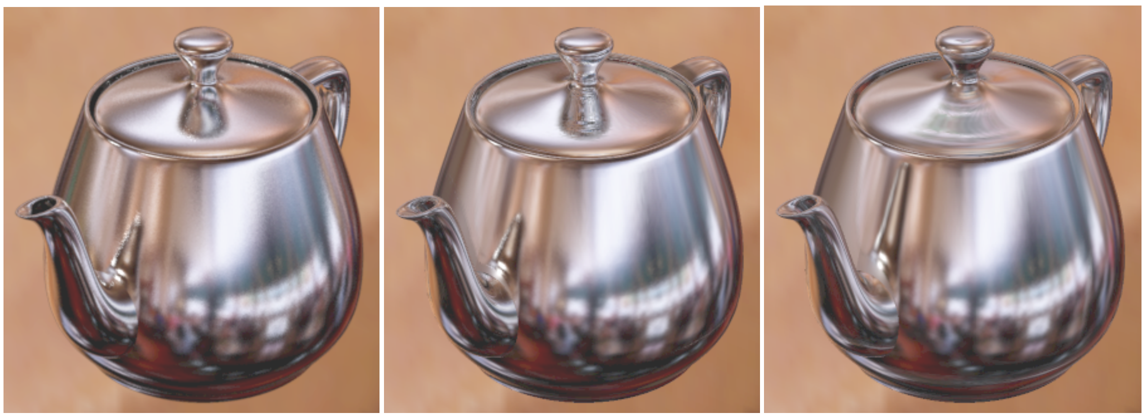


Figure 5.12: **Results Above Viewpoint Samples for a Glossy Teapot.**

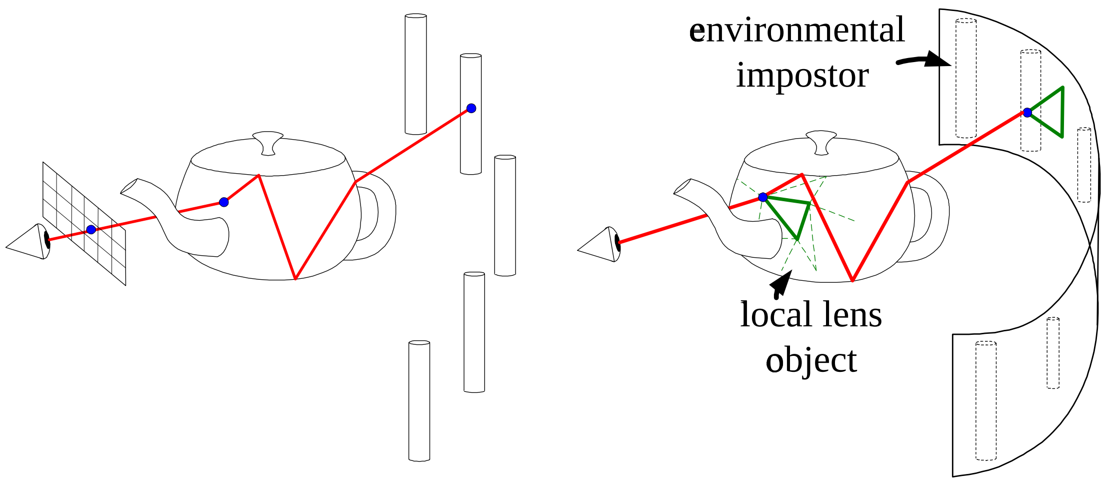
Chapter 6

Hybrid Rendering

In addition to being able to handle reflective objects away from the pre-rendered views, as we did in Chapter 5, we would like to be able to handle refractive objects. Unfortunately, with refractive objects it is difficult to predict the eventual outgoing ray direction based on what happens when a ray from the viewpoint first hits the refractive object. At the very least, the ray must interact with two surface interfaces, the interface from air to glass and glass to air before it leaves the object. For real objects, the ray may pass through several layers of glass before leaving the object. Our solution to this problem is *hybrid rendering.*

Recall that Z-buffer hardware is well-suited for rendering texture-mapped 3D geometry but inadequate for rendering reflective and refractive objects. It rasterizes geometry with respect to a constrained ray set – rays emanating from a point and passing through a uniformly parameterized rectangle in 3D. Reflective and refractive objects create a more general lens system mapping each incoming ray into a number of outgoing rays, according to a complex, spatially-varying set of multiple ray “bounces”. Environment maps (EMs) [Bli76] extend hardware to simulate reflections from an infinitely distant environment, but ignore all but the first bounce and so omit self-reflections and refractions. On the other hand, ray tracing [Whi80] generates these effects but is unaccelerated and incoherently accesses a large scene database. Nevertheless, modern CPUs are becoming powerful enough to perform limited ray tracing during real-time rendering.

In hybrid rendering, we combine the benefits of both systems by tracing ray paths through reflective/refractive objects to compute how the local geometry maps incoming rays to outgoing. To encapsulate more distant geometry, we use the outgoing rays as indices into a previously-inferred EM per object, allowing efficient access and resampling by graphics hardware.



(a) Ray-Tracing (b) Hybrid Rendering

Figure 6.1: **Comparison between Ray-Tracing and Hybrid Rendering.** In ray-tracing, rays pass through samples on the image plane and are intersected with an object, a glass teapot in this example. A single ray bounces around through the teapot, eventually exiting. It is then intersected with environmental geometry, in this case a ring of columns. The rendering is unaccelerated by hardware and incoherently accesses a large scene database. In hybrid rendering, the idea is to leverage graphics hardware as much as possible. Rays are still traced, but only at vertices of the polygonal mesh for the glass teapot, which we call the local lens object. When the ray exits the local lens object, it is intersected with a simple impostor for the environment, a texture-mapped cylindrical shell in this example. A triangle on the local lens object thus maps to some triangle in the impostor's texture map so we can use texture mapping to interpolate within the triangle interior. Vertex ray-tracing takes place on the host processor, but now accesses only the local lens object. This provides better memory coherence than traditional ray-tracing.

We begin by segmenting reflective/refractive scene geometry into a set of *local lens objects*. Typically each glass or shiny object forms a single local lens object, but multiple objects can be combined if they are close or one contains or surrounds another. Because rays are traced through a system of only one or a few objects, the working set is smaller and memory access more coherent than with traditional ray tracing. To make the approach practical, we also initially limit the number of ray casts to polygon vertices, adaptively shooting additional rays only where necessary. We also prune the ray tree (binary for refractive objects where an incoming ray striking an interface generates child reflection and refraction rays) ignoring all but a few ray paths that still approximate the full ray traced rendering well (see Figure 6.4).

Each local lens object's EM is different from traditional ones: it is *inferred*, *layered*, and *parameterized*. Inferred means our EMs are computed as a least-squares best match to a pre-computed, ray traced image at a viewpoint *when applied as an EM in a rendering by the target graphics system*. The alternative of sampling a spherical image of incident radiance at a point (typically the lens object's center) ignores view-dependent shading in the environment and produces a less accurate match at the viewpoint. Layered means that we use multiple environmental shells to better approximate the environment as introduced in Chapter 5. Parameterized means we compute an EM per viewpoint over a set of viewpoints (Figure 6.2) to provide view-dependent shading on imaged objects and reduce the number of layers needed for accurate parallax. Our examples use a 1D viewpoint subspace that obtains accurate results over that subspace and plausible results for any viewpoint.

Our contributions include the hybrid rendering shading model (Section 6.2) and its combination of dynamic ray tracing with hardware-supported EMs (Section 6.4). We improve on the parameterized EMs (PEMs) described in the previous chapter by generalizing to more than two layers and providing tools for determining their placement (Section 6.3). In addition, we handle self-reflections by ray tracing the local geometry rather than representing it as an EM, and so achieve good results with as much as ten times sparser sampling of EMs. The method of Chapter 5 also has the problem that its inferred EMs represent only the part of the environment imaged at one viewpoint, resulting in disocclusions from nearby viewpoints. We ensure completeness in the layered EMs by matching to a layered image that includes occluded surfaces, as well as simultaneously over multiple nearby viewpoints or direct images of the environment. Results show our method supports rendering of realistic reflective and refractive objects on current graphics hardware (Section 6.5).

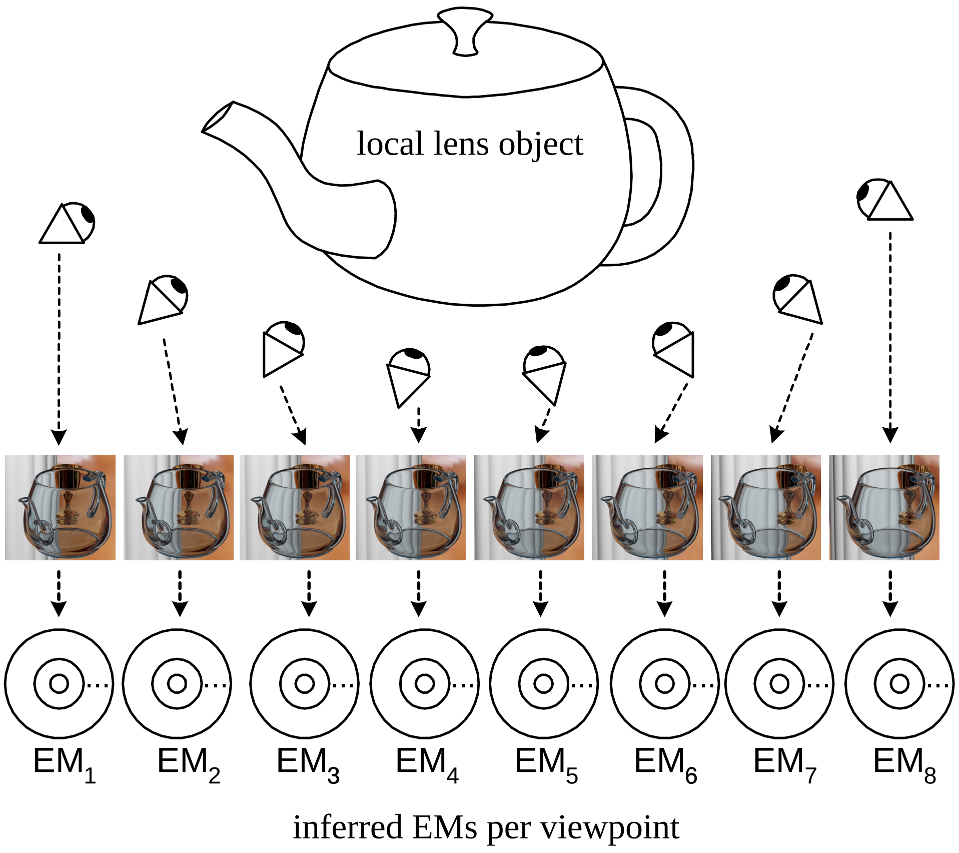


Figure 6.2: **Layered, Parameterized Environment Maps recorded over a set of viewpoints for each local lens object.** Each EM$_i$ consists of layered shells at various distances from the local lens object's center.

6.1 Comparison with Alternative Approaches for Refractions

Heidrich et al. [Hei99a] attempt to handle refractive as well as reflective objects using a light field to map incoming view rays into outgoing reflected or refracted rays. These outgoing rays then index either a static environment map, which ignores local effects further from the object, or another light field, which is more accurate but also more costly. Though our hybrid rendering similarly partitions local and distant geometry, we obtain sharper, more accurate reflections and refractions using local ray tracing rather than light field remapping. We also exploit the texture-mapping capability of graphics hardware using layered EMs for the more distant environment rather than light fields.

Chuang et. al. [Chu00] and Zongker et. al. [Zon99] capture the remapping of incident rays for real and synthetic reflective/refractive objects, but only for a fixed view. Kay and Greenberg [Kay79] simulate refractive objects with a simple, local model restricted to surfaces of uniform

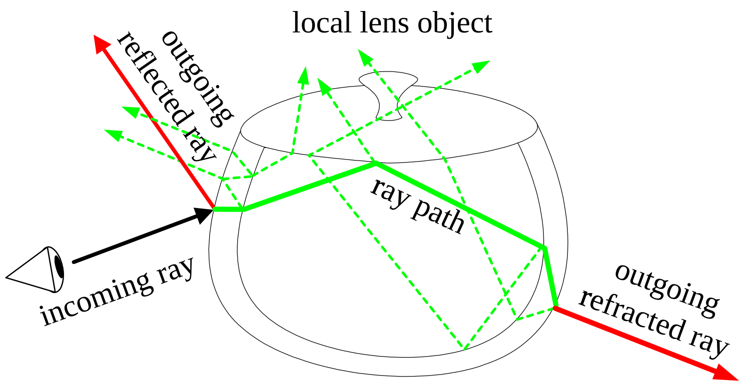


Figure 6.3: **Greedy Ray Path Shading Model.** When a ray from the viewpoint first strikes the lens object, we consider two paths: one beginning with an initial reflection and the other with an initial refraction. These paths are then propagated until they exit the local lens object by selecting the child ray having the greatest Fresnel coefficient. The dashed green ray paths represent paths that are ignored because the branching child ray had a smaller Fresnel coefficient than the solid green path.

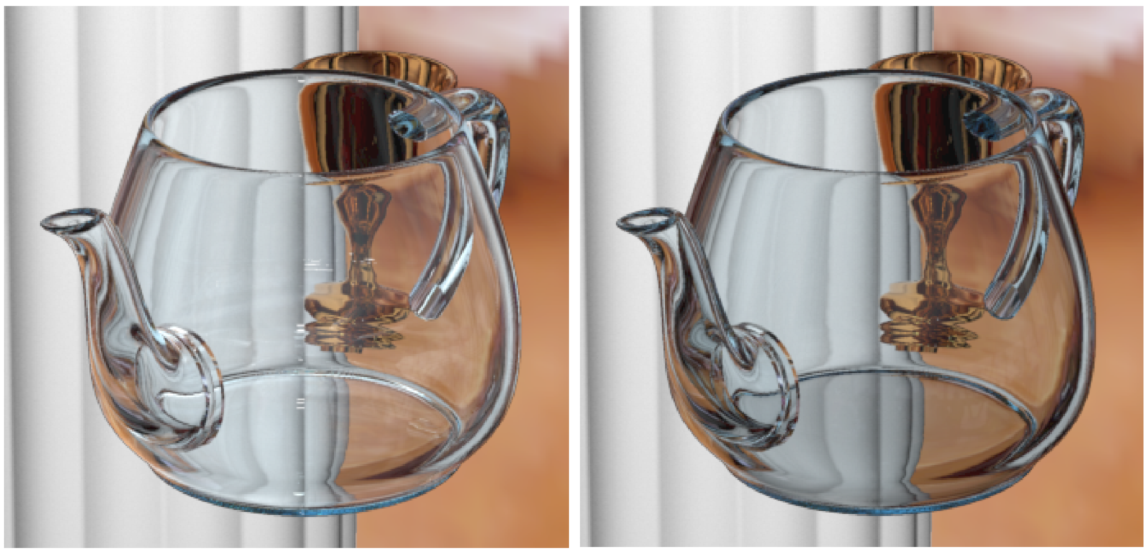
thickness.

Adaptive sampling has been used in ray tracing since it was first described [Whi80]. To decouple local and distant geometry, our adaptation is based on ray path, not color or radiance, differences. Kajiya's idea of ray paths rather than trees [Kaj86] forms the basis of our local model. Finally, the caching and ray intersection reordering of Pharr et. al. [Pha97] is another, completely software-based approach for memory-coherent access to the scene database.

6.2 Shading Model

Ray tracing simulates refractive objects by generating child reflective and refractive rays whenever a ray strikes a surface interface. The relative contribution of these children is governed by the Fresnel coefficients [Hec87], denoted $\hat{F}_R$ and $\hat{F}_T$ for reflected and transmitted (refracted) rays, respectively. These coefficients depend on the ray's angle of incidence with respect to the surface normal and the indices of refraction of the two media at the interface. Purely reflective objects are simpler, generating a single child reflected ray modulated by $\hat{F}_R$ but can be considered a special case with $\hat{F}_T = 0$.

Rather than generating a full binary tree for refractive objects which can easily extend to thousands of ray queries, we use a two-term model with greedy ray path propagation, illustrated in Figure 6.3. When a ray from the viewpoint first strikes the refractive object, we consider two paths:



(a) Full ray tree (b) Two-term greedy ray path

Figure 6.4: **Comparison of Shading Models.** The full ray tree (a) requires 5 times more ray queries than our greedy ray path model (b).

one beginning with an initial reflection and the other with an initial refraction. These paths are then propagated until they exit the local object by selecting the child ray having the greatest Fresnel coefficient.

The result is two terms whose sum approximates the full binary tree. Reflective objects require only a single term but use the same ray propagation strategy in case the local system contains other refractive objects. Figure 6.4 compares the quality of this approach with a full ray tree simulation.

Our model also includes a simple transparency attenuation factor, G, which modulates the ray color by a constant for each color channel raised to a power depending on the thickness of glass traversed between interfaces [Kay79]. The resulting model is

$$T\,F_T\,G_T \;+\; R\,F_R\,G_R$$

where respectively for the refracted and reflected ray paths: T and R are radiances along exit ray, F_T and F_R multiply the Fresnel coefficients along the path, and G_T and G_R multiply the transparency attenuation along the path (see Figure 6.5).

As a preprocess, for each viewpoint sample, we use a modified ray tracer to compute the two terms of this model as separate images. We then infer an EM that produces the best least-squares

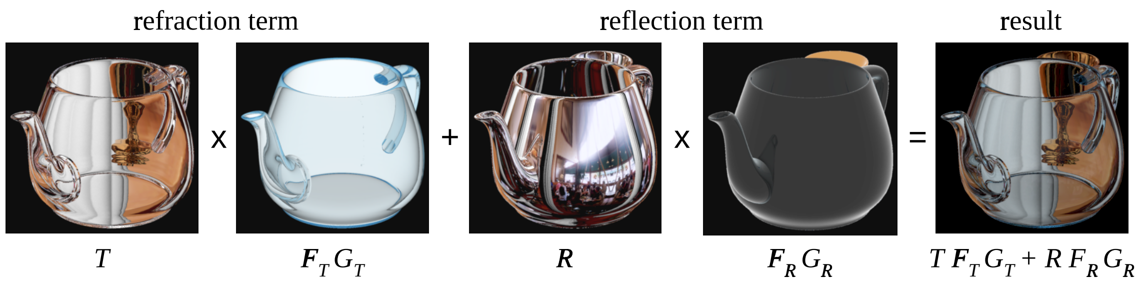


Figure 6.5: **Two-term Modulated Shading.**

match to both terms simultaneously. The result is a viewpoint-dependent sequence of inferred EMs. Each viewpoint's EM is layered by segmenting the environmental geometry into a series of spherical shells.

At run-time, we dynamically trace rays through each vertex of the local geometry according to our ray path model to see where they exit the local system and intersect the EM shells. We select the EM corresponding to the viewpoint closest to the current view or blend between the two closest. A separate pass is used for each term and then summed in the framebuffer. The Fresnel and transparency attenuation factors are accumulated on-the-fly as the path is traced, and produce per-vertex terms that are interpolated over each triangle to modulate the value retrieved from the EM. A better result can be achieved using 1D textures that tabulate highly nonlinear functions such as exponentiation [Hei99b].

While our current system neglects it, a diffuse component can be handled as an additional view-independent diffuse texture per object that is summed into the result. Such textures can be inferred to match ray traced imagery, as discussed in Chapter 3.

6.3 Layered Environment Maps

In Chapter 5, we introduced the idea of segmenting the environment into local and distant layers, and representing each layer in a separate environment map. In this section, we generalize to multiple environmental shells, as illustrated in Figure 6.6. A local lens object is associated with a layered EM per viewpoint in which each layer consists of a simple, textured surface. We use a nested series of spherical shells sharing a common origin for the geometry because spheres are easy to index and visibility sort. Other kinds of simple geometry, such as finite cubes, cylinders, and ellipsoids, may be substituted in cases where they more accurately match the actual geometry of the environment. A layer's texture is a 4-channel image with transparency, so that we can see through inner shells

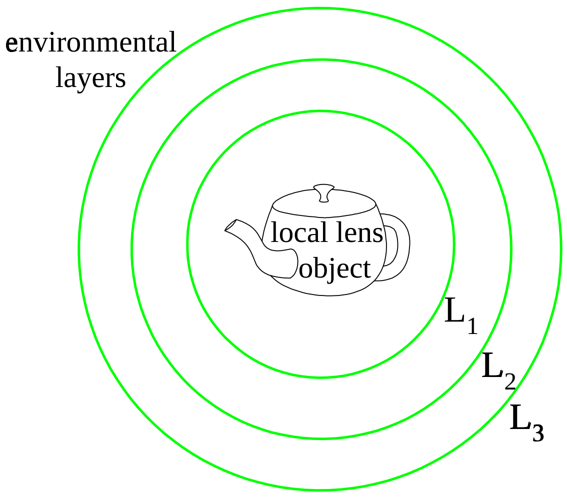


Figure 6.6: **Layered Environment Maps.** We use a series of multiple layers to approximate the environment. The nested spheres shown here are one example. This provides a better approximation than a single layer and still supports efficient hardware access. Layering the environment reduces conflicts where multiple points in the environment map to an identical point on the impostor. The result of such conflicts is blurriness or ghosting in the reflections and refractions. Layering also tends to increase the coherence in each of the layers,which is useful when compressing the resulting textured impostors. Finally, layering can represent parts of the environment that are occluded in a particular view. Each layer's image includes transparency so that we can see through it where it is partially occupied to more distant layers.

to outer ones where the inner geometry is absent. At run-time, we perform hybrid rendering to compute outgoing rays and where they intersect the layered EMs. We use the "over" blending mode to composite the layers L_i in order of increasing distance before modulating by the Fresnel and transparency attenuation terms, F and G, via

$$(L_1 \text{ over } L_2 \text{ over } \ldots \text{ over } L_n)\; F\; G$$

for each term in the two-term shading model.

6.3.1 Compiling the Outgoing Rays

To build layered EMs, the ray tracer compiles a list of intersections which record the eventual *outgoing rays* exiting the local lens object and where they hit the more distant environment. These intersections are generated from *incoming rays* originating from a supersampled image at a particular viewpoint including both terms of the shading model (reflection and refraction), each of which generates different outgoing rays (Figure 6.7a). Between incoming and outgoing rays, the ray paths are propagated using the model of Section 6.2. The intersection record also stores the image position of the incoming ray and color of the environment at the outgoing ray's intersection.

To avoid disocclusions in the environment as the view changes, we modified the ray tracer to continue rays through objects to reach occluded geometry. For each outgoing ray, we record all front-facing intersections with environmental geometry along the ray, not just the first (Figure 6.7c). Once the layer partitions are computed (Section 6.3.2), we then discard all but the first intersection of each outgoing ray with that layer. This allows reconstruction of parts of the environment in a distant layer that are occluded by a closer one. We also continue incoming rays in a similar fashion (Figure 6.7b) so that occluded parts of the lens object still generate intersection records. For example, we continue incoming rays through a cup to reach a glass teapot it occludes.

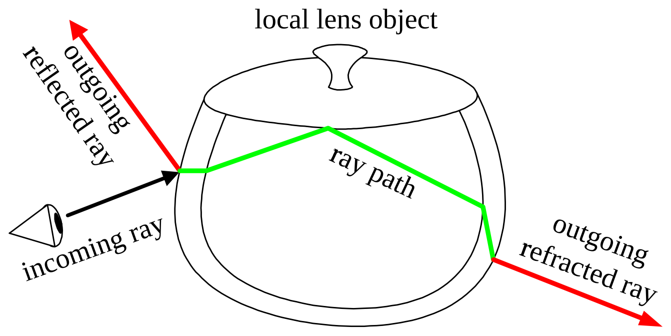
6.3.2 Building Layered EM Geometry

To speed run-time performance, we seek a minimum number of layers. But to approximate the environmental geometry well, we must use enough shells and put them in the right place.

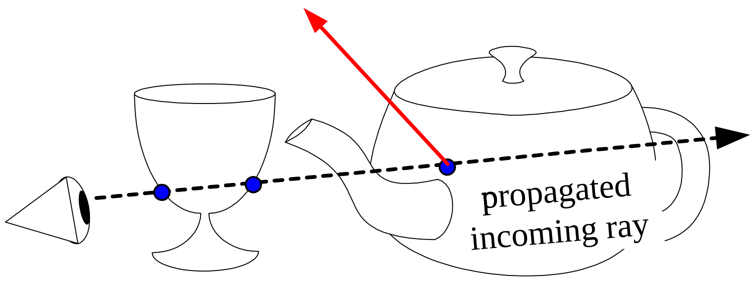
We use the LBG algorithm [Lin80] developed for compression to build vector quantization codebooks. The desired number of layers is given as input and the cluster origin is computed as the centroid of the local object. The LBG algorithm is run over the list of intersections, clustering based on distance to this origin. This algorithm begins with an initial, random set of cluster distances and assigns each intersection to its closest cluster. It then recomputes the average distance in each cluster, and iterates the assignment of intersection to closest cluster. Iteration terminates when no intersections are reassigned.

When partitioning geometry into layers, parts of coherent objects should not be assigned to different layers. This can cause incorrect tears in the object's reflected or refracted image. Our solution assigns whole objects only to the single cluster having the most intersection records with that object. The clustering algorithm should also be coherent across the parameterized viewpoints. This is accomplished by clustering with respect to all viewpoints simultaneously. Figure 6.11 shows clustering results on an example scene in which our algorithm automatically segments a glass teapot's environment into three layers.

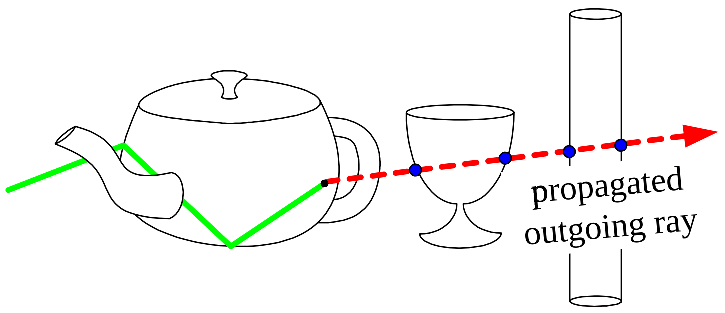
Layer shells are placed at the average distance of intersections in the cluster, where "continued ray" intersections are represented only by their frontmost cluster member.



(a) Incoming/outgoing rays



(b) Incoming ray propagation



(c) Outgoing ray propagation

Figure 6.7: **Incoming/Outgoing Rays and Ray Propagation.**

6.3.2.1 Layer Quads

Often a spherical shell is only sparsely occupied. In that case, to conserve texture memory we use a simple quadrilateral impostor for this geometry rather than a spherical one. To define the impostor quadrilateral, we can find the least-squares fit of a plane to the layer's intersection points. A simpler, but less optimal, scheme is to find the centroid of the layer's intersection points and define the normal of the plane as the direction from the lens object center to the centroid. The plane's extent is determined from a rectangular bounding box around points computed by intersecting the outgoing rays associated with the layer's intersection records with the impostor plane. One complication is that during run-time, outgoing rays may travel away from the quad's plane, failing to intersect it. This results in an undefined texture access location. We enforce intersection for such rays by subtracting the component of the ray direction normal to the impostor plane, keeping the tangential component but scaling it to be very far from the impostor center.

Texture maps for spherical shells or quads are computed using the same method, described below.

6.3.3 Inferring Layered EM Texture Maps

We use the least-squares inference approach described in Chapter 3, and apply it in the same manner as in Chapter 5. In particular, when computing an EM the rendering matrix is created using test renderings with environment mapping instead of texture mapping.

As discussed in the last chapter, there are two advantages of this inference method over a simple projection of the environment onto a series of shells. By matching a view-dependent ray traced image, it reproduces view-dependent shading on reflected or refracted objects, like a reflective cup seen through a glass teapot. It also adjusts the EM to account for the geometric error of approximating environmental objects as simpler geometry, such as a spherical shell. The result is better fidelity at the ray traced viewpoint samples.

We infer each layer of the layered EM independently, but simultaneously over both terms of the shading model (Figure 6.11). After performing the layer cluster algorithm on samples from a supersampled image, each layer's samples are recombined into a single image and filtered to display resolution to form two images b_R and b_T, corresponding to the two terms of the shading model. Only a single image is needed for reflective objects. These images have four channels – the alpha channel encodes the fraction of supersampled rays through a given pixel whose outgoing ray hit environmental geometry in that layer, while the rgb channels store the color of the environment intersected by those outgoing rays. We infer the two rendering matrices, A_R and A_T, corresponding respectively to hybrid rendering (Section 6.4) with an initial reflection or refraction. We then find the least-squares solution to

$$\begin{aligned} A_R\, x &= b_R \\ A_T\, x &= b_T \end{aligned} \tag{6.1}$$

to produce a single EM for the layer, x, matching both terms. Figure 6.11 shows an example of these b terms and resulting inferred EM, x, for each of three layers.

It is possible for the ray tracer to generate widely diverging rays when sampling a single output pixel, causing noise in the environment map solution. We therefore modified the ray tracer to generate a confidence image. The per-pixel confidence value is computed as a function of the maximum angle between the directions of all ray pairs contributing to the particular pixel. More precisely, we use the formula

$$1 - \min(\theta_m^2, \theta_c^2)/\theta_c^2$$

where θ_m is the maximum angle between ray pairs and $\theta_c = 5^\circ$ is an angular threshold. We multiply the confidence image with both sides of Equation (6.1) prior to solving for x.

To conserve texture memory, it is beneficial to share more distant EM layers between local lens objects. To do this, we can add more equations to the linear system (6.1) corresponding to multiple lens objects and simultaneously solving for a single EM x.

As observed in Chapter 5, choosing the proper EM resolution is important to preserve frequency content in the imaged environment. A very conservative approach generates test renderings to determine the most detailed EM MIPMAP level actually accessed by the graphics system. Texture memory bandwidth and capacity limitations may dictate the use of lower resolutions.

6.3.3.1 Simultaneous Inference Over Multiple Viewpoints

A difficulty with this inference technique is that the lens objects can fail to image all of its environment. For example, a flat mirror does not image geometry behind it and a specular fragment images only a small part of its environment from a particular viewpoint. These missing portions can appear in a view near but not exactly at the pre-rendered viewpoint. We solve this problem by inferring EMs that simultaneously match ray traced images from multiple views. The views can be selected as a uniform sampling of a desired viewspace centered at the viewpoint sample.

To compute simultaneous viewpoint inference, outgoing rays are compiled for each of these multiple viewpoints. A single EM layer, x, is then inferred as a least-squares simultaneous match at all viewpoints, using the system of equations (6.1) for each viewpoint. Although this blurs view-dependent shading in the environment, good results are achieved if the set of viewpoints matched are sufficiently close.

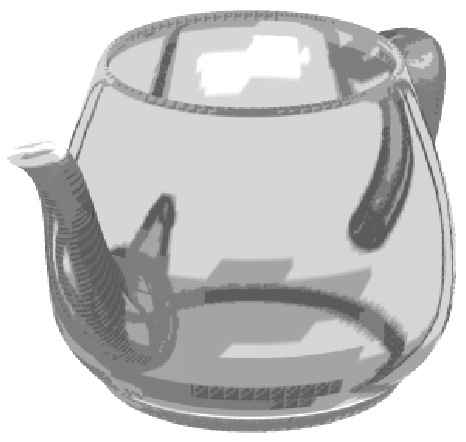


Figure 6.8: **Adaptive Tessellation Visualization.**

An alternative method to fill in the missing portions in the environment map is to infer it using extra rays in addition to the ones that exit the lens object. These additional rays can be taken from the object center as in traditional environment maps. They can also be taken from the viewpoint, but looking directly at the environment (i.e., without the lens object), to approximate view-dependent shading in the environment. Direct images from the lens object center tend to work better for reflective objects while direct images from the viewpoint are better suited for refractive (transparent) objects.

We use the confidence-weighted, least-squares solution method in (6.1), but solve simultaneously across images of the lens object from the viewpoint as before (*lens object images*), combined with direct images of the environment without the lens object (*direct images*). In these direct images, per-pixel confidence is computed as a function of the likelihood that the pixel represents a missing portion of the environment. We compute this via distance of the direct image ray's intersection with its closest point in the intersection records of the lens object images. The advantage of using direct images is that it is possible to fill in the missing portions of the environment with a fixed, small number of extra images (typically 6 when rendering from the object center), regardless of the size of the viewspace around a viewpoint sample.

6.4 Hybrid Rendering

6.4.1 Adaptive Tessellation

Performing ray tracing only at lens object vertices can miss important ray path changes occurring between samples, producing very different outgoing rays even at different vertices of the same triangle. Our solution is to perform adaptive tessellation on the lens object based on two criteria: the ray path “topology” and a threshold distance between outgoing rays. Using topology, ray paths are considered different if their path lengths are different, or the maximum coefficient at an interface changes between reflection and refraction. Using outgoing ray distance, they are different if the angle between their directions is more than $3°$. Figure 6.8 illustrates adaptive tessellation using an example image shaded darker with increasing subdivision level. Places where rays go through more interfaces or where the surface is highly curved require more sampling.

When the ray paths at a triangle’s vertices are too different, the triangle is subdivided at the midpoints of each of its edges in a 1-to-4 subdivision, and the metric is recursively applied. The process is continued until the three ray paths are no longer different or the triangle’s screen-projected edge lengths are less than a threshold, τ. We also allow 1-to-3 and 1-to-2 subdivision in cases where some of the triangle edges are already small enough (see Figure 6.9). We adapt the tessellation simultaneously for both terms of the shading model, subdividing a triangle if either ray path is considered different. We ignore differences in subdivision between neighboring triangles, fixing the resulting “t-junction” tessellation as a postprocess. A hash table on edges (vertex pairs) returns the edge midpoint if it has already been computed from a more highly subdivided neighboring triangle. Recursive querying yields all vertices along the edge.

Given all the boundary vertices, it is simple to compute a triangular tessellation that avoids t-junctions. The hash table is also used to quickly determine whether a triangle vertex ray has already been computed, thus avoiding redundant ray queries.

To avoid unnecessary ray tracing and tessellation in occluded regions, we compute whether each vertex is directly visible from the viewpoint using a ray query. If all three vertices of a triangle are occluded, we do not subdivide the triangle further, but still compute correct texture coordinates for its vertices via ray tracing in case some part of its interior is visible. Another optimization is to avoid ray tracing at vertices whose triangles are all backfacing. Using a pass through the faces, we mark each triangle’s vertices as “to be ray traced” if the triangle is front-facing; unmarked vertices are not ray traced.

6.4.2 Multipass Rendering with Layered EM Indexing

Assuming we have a refractive object with n layers in its EM, we perform n passes for its refractive term, and n passes for the its reflective term. We multiply each term by the $F\ G$ function interpolated by the graphics hardware across each triangle. The texture index for each term/layer pass is generated by intersecting the outgoing ray with the layer’s geometric impostor, such as a sphere. Taking this point of intersection and subtracting the EM origin point yields a vector that forms the hardware EM index, recorded with the vertex. As in Chapter 5, we use a hardware-supported cube

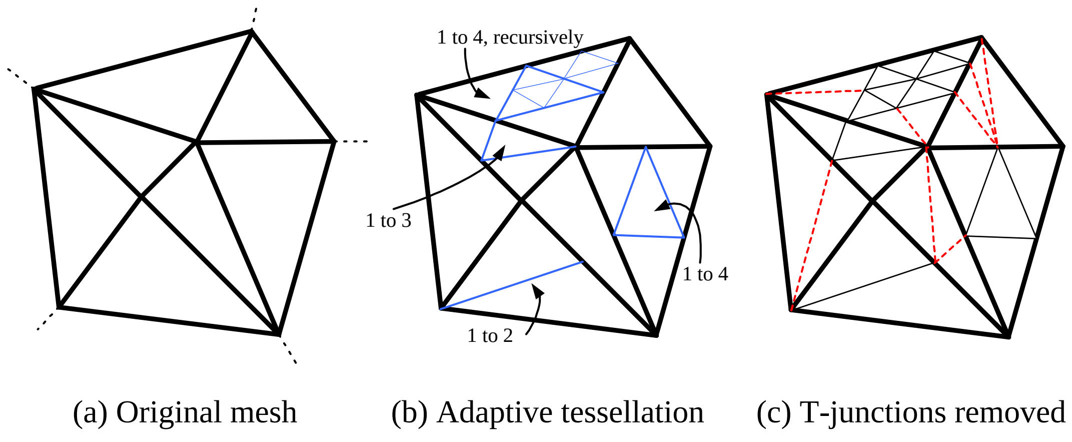


Figure 6.9: **Adaptive Tessellation and T-junction removal.** The original mesh, (a), is adaptively subdivided when ray paths at the vertices are sufficiently different, (b). The resulting t-junctions are removed by additional tessellation of adjacent triangles, (c), illustrated with dotted lines, to form a triangular tessellation.

map spherical parameterization which doesn't require normalization of this vector. Note that the texture indices change per layer since the distances to the EM spheres are different and the outgoing rays do not emanate from the EM origin.

When rendering from a viewpoint away from a pre-rendered sample, a smoother result is obtained by interpolating between the two closest viewpoints. Thus, we perform $4n$ passes, $2n$ for each viewpoint, blended by the relative distance to each viewpoint. Ray tracing, adaptive tessellation, and texture coordinate computation are performed just once per frame. Ray tracing is performed with respect to the actual viewpoint, not from the adjacent viewpoint samples.

Each layer is computed in a separate pass because of texture pipeline limitations in the current graphics system (Microsoft Direct3D 7.0 running on an Nvidia GeForce graphics accelerator). To begin the series of compositing passes for the second of the two summed shading terms, the framebuffer's alpha channel must be cleared. This is accomplished by rendering a polygon that multiplies the framebuffer's rgb values by 1 and its alpha by 0. We then render its layers from front to back, which sums its contribution to the result of the first term's passes. With the advent of programmable shading in inexpensive PC graphics hardware and the ability to do four simultaneous texture accesses in each pass, it will be possible to reduce those $4n$ passes to n and avoid the alpha clear step.

6.5 Results

We tested a scene containing a glass teapot, a reflective cup, a ring of columns, and distant walls. EMs were parameterized by viewpoints circling around the teapot in 8° increments. We inferred EMs from 5 viewpoints simultaneously, uniformly distributed in a 7° range above and below the viewpoint plane. The scene contains two lens objects, a teapot and cup; we used our clustering algorithm to select 3 EM layers for the teapot and 2 for the cup. A quadrilateral impostor was used for the sparsely-occupied cup environmental layer of the teapot (Figure 6.11, top), a cylindrical shell for the columns environmental layer of the teapot (Figure 6.11, middle), and spherical shells were used for all other layers. We also tried a solution that was constrained to a single EM layer for each lens object, still using the clustering algorithm to determine placement of the single shell.

Figure 6.12 compares the quality of our results for two novel views: one in the plane of the circle of viewpoints (a), and one above this plane (b). Using multiple EM layers, we achieve quality comparable to the ray tracer. Reconstruction using a single layer is noticeably blurry because of conflicts where different points in the environment map to identical ones in the spherical shell approximation. Moreover, the interactive rendering results for the single layer solution show significant “popping” when switching between viewpoint samples. The multi-layer solution better approximates the environment, providing smooth transitions between viewpoints.

Together, we call the method of ray continuation to reach occluded geometry from Section 6.3.1, and the simultaneous solution across multiple viewpoints from Section 6.3.3, *EM disocclusion prevention.* Figure 6.10 shows the effectiveness of these methods in eliminating environmental disocclusions which would be obvious otherwise.

	$\tau = 3$	$\tau = 5$	$\tau = 10$
Ray tracing at vertices	13.48	7.82	4.40
Texture coordinate generation	0.71	0.38	0.21
Tessellation	2.11	0.83	0.26
Other (including rendering)	2.57	1.45	0.87
Total frame time	18.87	10.48	5.74
Time reduction factor	25.4	45.8	83.6
Ray count	1,023,876	570,481	298,684
Ray reduction factor	6	10.8	20.6
Triangle intersection tests	11,878,133	6,543,993	3,603,034
Intersection reduction factor	14.2	25.7	46.7

Table 6.1: Hybrid Rendering Runtime Performance

To measure performance, we tried three different adaptive subdivision thresholds of $\tau = 3$,

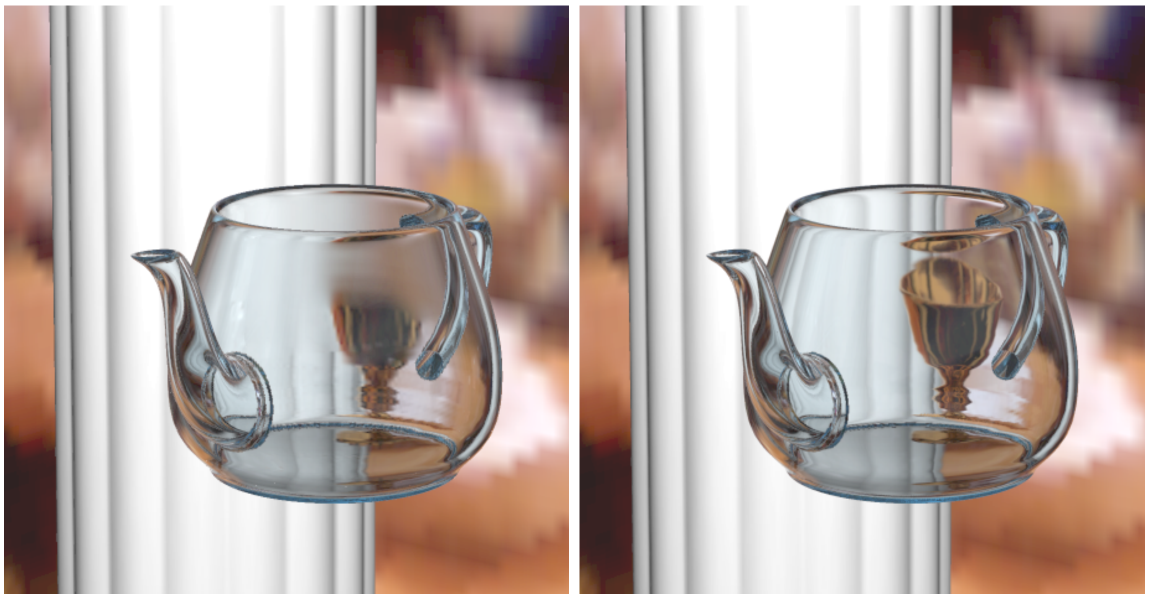

(a) Without prevention (b) With prevention

Figure 6.10: **EM Disocclusion.**

$\tau = 5$ and $\tau = 10$ (measured in subpixels) in a 3×3 subsampled 640×480 resolution rendering (see Table 6.1). Performance was measured in seconds on a 1080MHz AMD Athlon with Nvidia GeForce graphics card; reduction factors are with respect to ray tracing without hybrid rendering. For comparison, the ray tracer required 480 seconds to render each frame, using 6,153,735 rays and 168,313,768 triangle intersection tests. The version with $\tau = 3$ is shown in Figure 6.12; the three versions are compared in Figure 6.13. The faster $\tau = 5$ achieves good quality but suffers from some artifacts when animated. The difference is discernible in the still image in Figure 6.12 as slightly increased noise along edges such as the bottom of the teapot and where the spout joins the body. At $\tau = 10$ with approximately another factor of 2 reduction in rendering time, the edges become jagged because of coarse tessellation, resulting in low quality. As we relax the adaptive subdivision threshold, the artifacts are first seen along edges because that is where ray path changes are likely to occur.

Hybrid rendering was 25-84 times faster than a uniformly-sampled ray tracing, though both used identical ray casting code and the greedy ray path shading model. (Using a full tree shading model would incur an additional factor of 5.) This is not entirely accounted for by the reduction of roughly a factor of 6-21 in ray queries or 14-47 in triangle intersection tests obtained by hybrid rendering.

The reason for our increased performance is the increased locality we achieve by ray tracing only through the lens object's geometry, and the hardware acceleration of texture map access. Although adaptive sampling reduces triangle intersection tests and ray queries by roughly the same factor, triangle intersection tests (which include ray misses as well as actual intersections) are reduced by an additional factor because the environment is approximated with simple spherical shells.

Though our performance falls short of real-time, significant opportunity remains both to optimize the software and parameters (like the initial lens object's tessellation), and to tradeoff greater approximation error for higher speed. We note that more complicated environmental geometry will increase the benefit of our use of approximating shells. To speed up ray tracing, it may be advantageous to exploit spatial and temporal coherence, possibly combined with the use of higher-order surfaces rather than memory-inefficient polygonal tessellations. In any case, we believe that future improvement to CPU speeds and especially support for ray tracing in graphics hardware will make this approach ideal for real-time rendering of realistic shiny and glass objects.

6.5.1 Results for Offline Rendering Acceleration

One possible alternative application of hybrid rendering is in speeding up the offline rendering of realistic animations. Hybrid rendering could be used to interpolate between ray traced key frames at which view-dependent layered environment maps are inferred. Table 6.2 compares the cost of standard ray-tracing of an example 500 frame animation sequence with one that uses hybrid rendering. In making this comparison, we account for all the costs of hybrid rendering, including the

	Single frame	**Sequence multiplier**	**500 frames**
Standard ray-tracing	8 min.	500 frames	4000 min.
Ray-tracing with incoming/outgoing ray propagation	25 min.	7 key frames $\times$ 5 viewpoints	875 min.
Inferring environment maps simultaneously from 5 viewpoints	60 min.	7 key frames	420 min.
Hybrid rendering runtime	19 sec.	500 frames	160 min.
Total time for hybrid rendering			1455 min.
Time reduction factor			2.7

Table 6.2: Cost of ray-tracing vs. hybrid rendering for offline computation of a 500 frame realistic animation.

pre-processing costs. In particular, there is the added cost of incoming/outgoing ray propagation at the ray-traced key frames, the cost of performing ray-tracing at multiple viewpoints per key frame to handle disocclusions (5 viewpoints/key frame in our example), and the cost of inferring environment maps at each key frame. We assume 7 key frames are needed for hybrid rendering based on our experience in faithfully reproducing the entire 500 frame animation. In this comparison, we find that hybrid rendering is 2.7 times faster than standard ray-tracing using the same greedy ray path shading model. We conclude that using hybrid rendering to accelerate the offline rendering of realistic animations is a promising area for further study.

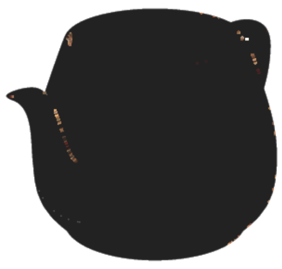
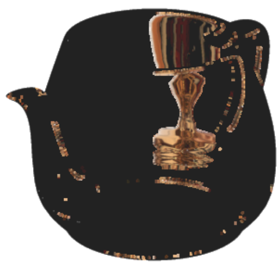
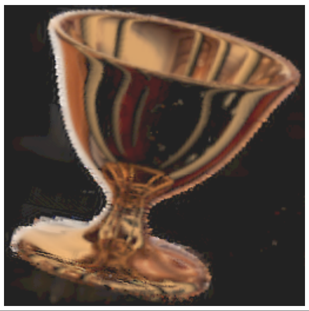
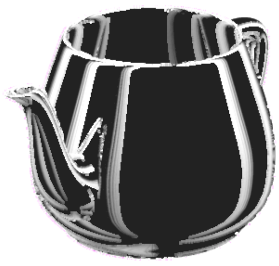
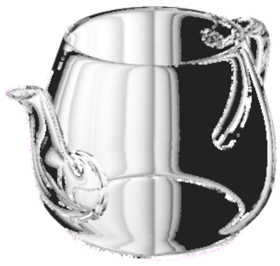
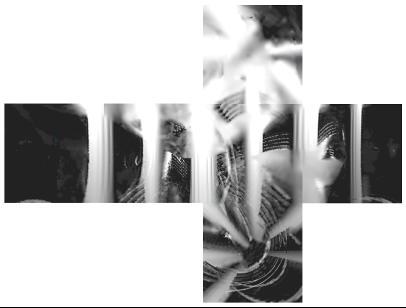
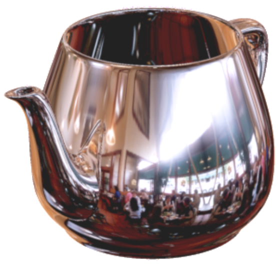
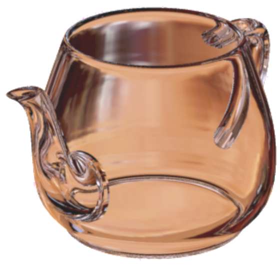
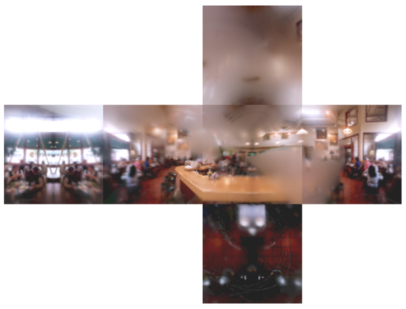
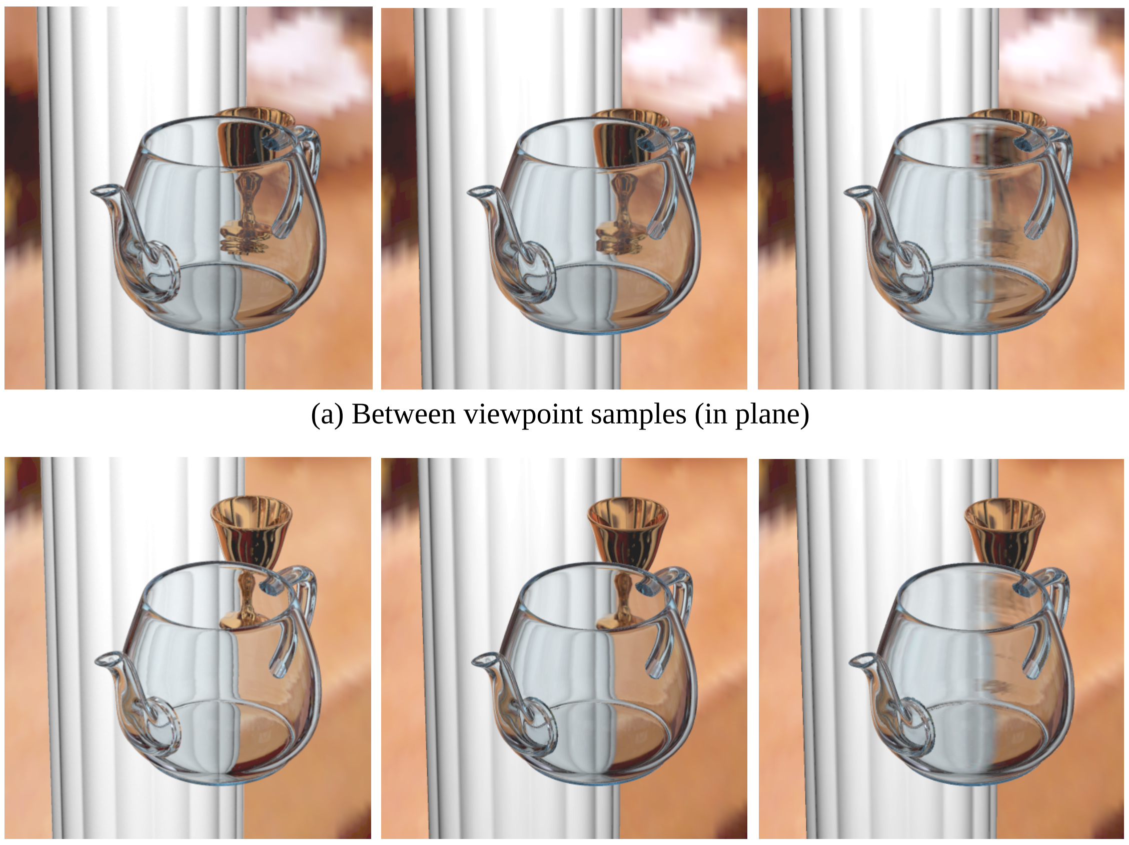
L_1 (cup)			
L_2 (cols)			
L_3 (walls)			
Layer	Reflection term $\boldsymbol{b}_R$	Refraction term $\boldsymbol{b}_T$	Inferred EM $\boldsymbol{x}$ from both terms

Figure 6.11: **Layered EMs inferred at a viewpoint sample for a glass teapot (three layers).** A quadrilateral was used for the L_1 layer, a cylindrical shell for L_2, and a spherical shell for L_3. Shells are parameterized by a six-faced cube map. Entire MIPMAPs are inferred; only the finest level is shown.

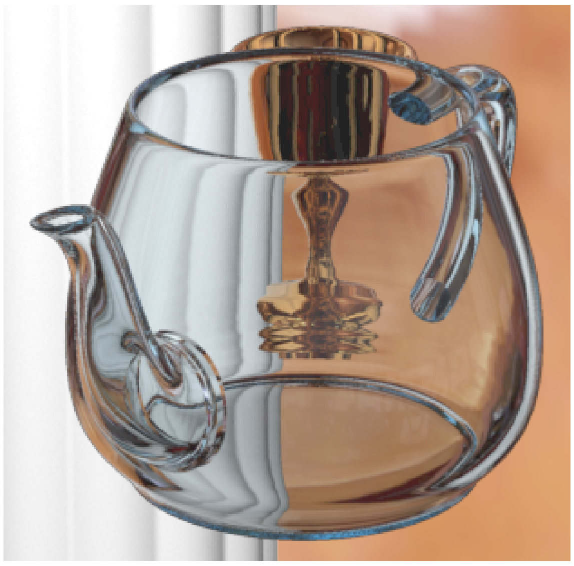


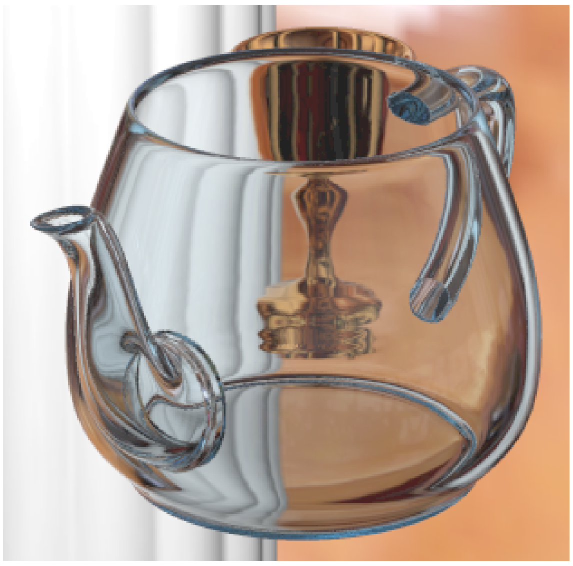
(a) Between viewpoint samples (in plane)

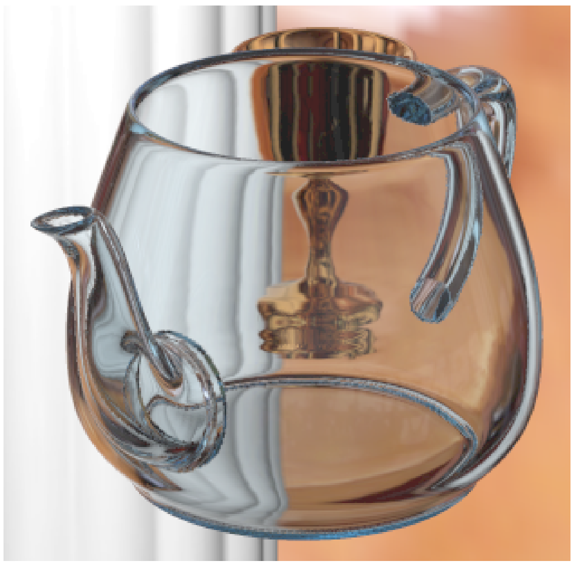
(b) Above viewpoint samples

Ray-Traced Hybrid (multi-layer) Hybrid (single layer)

Figure 6.12: **Hybrid rendering results.** The right two columns were generated by a PC graphics card.


(a) Ray-Traced (480 sec/frame)


(b) Hybrid *tau* = 3 (18.9 sec/frame)


(c) Hybrid *tau* = 5 (10.5 sec/frame)

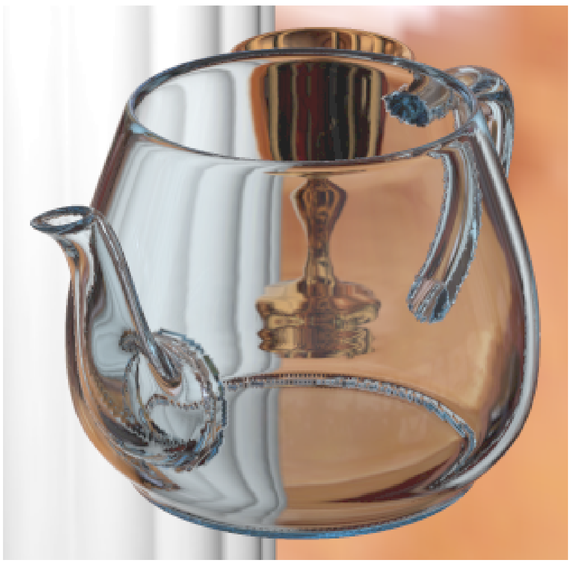

(d) Hybrid *tau* = 10 (5.7 sec/frame)

Figure 6.13: **Performance/Quality Tradeoff with Hybrid Rendering.**

Chapter 7

Conclusion

7.1 Summary and Conclusions

Applications like entertainment, human-computer interfaces, scientific visualization, and engineering have increasingly required more accurate simulation of physical processes that are visually perceptible. Physical simulation methods are by nature computationally intensive, and suitable only for offline rendering. At the same time, interactive graphics hardware that is optimized for rendering texture-mapped 3D geometry has continued to get cheaper and faster. However, such hardware, based on the standard *rendering pipeline*, is inherently inadequate for simulating all the effects that we might want in an interactive rendering, such as indirect lighting with reflections, refractions, and shadows. In our work, we have strived to bridge the gap between realistic image synthesis techniques and interactive rendering with the traditional graphics pipeline. We proposed a three-step model for achieving this goal. In the first step, we parameterize the multi-dimensional space, and render the images composing the space using a high-quality offline renderer. In the second step, we compress the parameterized image space efficiently by encoding the images in terms of primitives supported by the graphics pipeline and exploiting multi-dimensional coherence. Finally, in the third step, we treat graphics hardware as a decoder, and take full advantage of its fast rendering capabilities to reconstruct the original high-quality images.

Texture Inference by Inverse Rendering

The drastic improvements in performance and cost of the graphics pipeline in recent years has enabled inexpensive PC graphics hardware to perform flexible and fast texture mapping of polygons. We exploit this texture mapping capability throughout this thesis by transforming the high-quality offline rendered images into an alternative representation with texture and environment maps. When performing an *inverse rendering* conversion of this kind, it can be challenging to preserve fidelity to the original images. We have developed a novel method for texture inference to best match the graphics hardware rendering to the input ray-traced imagery.

Our inference approach is based on the observation that a texture pixel contributes to zero or more display pixels. Neglecting hardware quantization effects, a texel that is twice as bright contributes twice as much. Hardware rendering can thus be modeled as a linear system, called the *rendering matrix*, mapping texels to display pixels. To find the rendering matrix, we perform test renderings on the graphics hardware that isolate each texel's display contribution. Given the rendering matrix, A, we then find the least-squares best texture, x, which when applied matches the ray tracer's image, b. This results in the linear system $Ax = b$.

We compared our texture inference method with two alternative approaches: a "forward mapping" method in which texture samples are mapped to the object's image layer and interpolated using a high-quality filter, and a method that obviates the need for an intermediate image representation by rendering directly into textures. We find that these alternative methods produce a blurry and inaccurate result because they do not account for how graphics hardware filters the textures. In comparison, our least-squares optimization method achieves a 2.4-3.4 dB improvement in peak signal to noise ratio when rendering with an anisotropic filtering model in hardware. In addition, a major benefit of our optimization approach is that we can place additional constraints on the solution. In particular, we use a pyramidal regularization term to ensure smoothness in the resulting texture image, even in regions of the texture that are not used. Finally, it is worth mentioning that our least-squares texture inference approach generalizes easily to environment maps. Environment maps are inherently more difficult to solve for than texture maps because a one-to-one mapping between the object surface and the texture image cannot be guaranteed under all viewing conditions as with texture maps. In our solution, we apply the same method for inferring environment maps as we do for texture maps, with the only difference being that the rendering matrix is created using test renderings with environment mapping instead of texture mapping.

One may ask what is the "asymptotic" value of our inverse rendering approach as graphics filtering hardware improves in the future. Before answering this question, we note that high quality filtering will always be a challenge for graphics hardware because of the large kernel support that a high quality filter would entail. Current hardware relies on pre-filtered MIPMAP pyramids [Wil83], and is capable of performing either isotropic trilinear filtering or anisotropic filtering with limited anisotropy factors. Assuming high-quality texture filtering is free, we expect that as graphics filtering hardware becomes closer to ideal, the approach of rendering directly into texture maps would become more competitive with our inverse fitting approach. However, from a pre-processing efficiency standpoint, rendering into images with the accompanying inverse rendering step will in most cases be measurably less costly than rendering directly into textures. We are primarily interested in the final image that appears on the screen. Rendering into images has the advantage of placing samples where they contribute the most to image quality, and so we can achieve equivalent image quality to rendering directly into texture maps with a fewer total number of samples.

Another related question is what is the value of highly optimized inverse rendering if at run-time we move away from the exact image samples where inverse fitting is performed. Moving away from pre-rendered viewpoints inherently reduces the usefulness of our inverse fitting approach because inverse fitting only compensates for the hardware filter model at the exact viewpoint samples where we match to the input ray-traced imagery. Still, we can imagine applications where we always stay at the pre-rendered samples. Finally, as an answer to both questions we have raised above, we believe that an inverse fitting method will always be needed to extract textures from images in situations where we do not have access to the inner workings of the system generating the imagery. Note that most high quality renderers in existence today produce images as output, not textures. Ideally, we would like our overall system to be able to robustly work with any high-quality image renderer, without requiring any modifications to the rendering code. Inferring textures from images independently of the rendering system used to produce the images is thus a significant benefit. Although we have focussed on synthetic imagery throughout this thesis, we can also imagine our inverse rendering approach being applied to real-world images captured using real cameras when combined with computer vision techniques for inferring object geometry.

Parameterized Texture Compression

Inferring a separate texture map for an object at each point in the image space results in a parameterized texture. In our work on compressing parameterized textures, we exploit coherence between texture images through the use of an adaptive Laplacian pyramid. The adaptive nature of this pyramid recognizes that not all parameters in an image space are equally coherent, as is often assumed in image coders. We also exploit coherence that is present within a single texture image through the use of standard 2D compression methods such as JPEG and wavelet-based SPIHT encoding. To enhance compression, we separate the diffuse and specular lighting layers. We also employ automatic bit allocation methods from signal compression to allocate storage across levels within a Laplacian pyramid, across lighting layers, and across objects. While alternative compression schemes could have been used to compress the parameterized textures, we picked the Laplacian pyramid approach because it avoids processing large fractions of the representation to decode a given texture image, thus enabling fast decoding at run-time.

Image-based rendering (IBR) and other techniques compress images of *views*. We compress textures instead. There are three benefits to compressing parameterized textures instead of parameterized images produced by the high-quality renderer. First, encoding a separate texture map for each object better captures its coherence across the parameter space independently of where in the image it appears. This is especially the case for diffusely shaded objects. Second, object silhouettes are correctly rendered from actual geometry. Because the silhouettes are not encoded in image form, they suffer fewer compression artifacts. Finally, we can move the viewpoint away from the original parameter samples without revealing geometric disocclusions.

Results for parameterized textures show that quite good quality can be achieved at a compression factor of 384:1 and reasonable quality at 768:1. Unlike previous work in multi-dimensional IBR, we also show our methods to be superior to a state of the art image sequence coder applied to a sensible collapse of the space into 1D. Specifically, comparing MPEG4 encoding of parameterized images at 355:1 with adaptive Laplacian encoding of parameterized textures at 379:1, we achieve better perceptual quality, especially along object silhouettes. Empirically, we obtain a 1.9 dB improvement in peak signal to noise ratio. Our run-time achieves near real-time (˜5Hz) decoding on graphics hardware with full resolution textures, and faster real-time (˜31Hz) decoding at reduced texture resolution. In summary, we have demonstrated a complete system that achieves high quality rendering of parameterized image spaces at compression ratios up to 800:1 with interactive playback on current consumer graphics cards.

Generalized Multi-dimensional Image Spaces

One novel aspect of our work is that we generalize to image spaces with arbitrary parameters, not just viewpoint or time. We are free to parameterize the radiance field based on time, position of lights or viewpoint, surface reflectance properties, object positions, or any other degrees of freedom in the scene, resulting in an arbitrary-dimensional parameterized image space. To compile such spaces, an image at each point in the parameter space must be rendered, so compilation is exponential in dimension. We believe our compilation approach is good for spaces in which all but one or two dimensions are "secondary"; i.e., having relatively few samples. Examples include viewpoint movement along a 1D trajectory with limited side-to-side movement, viewpoint changes with limited, periodic motion of some scene components, or time or viewpoint changes coupled with limited changes to the lighting environment.

Parameterized Environment Maps

One of the drawbacks of capturing shading effects as parameterized texture maps (PTMs) is that we get a pasted on look for reflective objects when we move our viewpoint away from the prerendered views. Traditional environment maps have been developed in graphics for handling reflective objects, but fail to capture local reflections including effects like self-reflections and parallax in the reflected imagery. Image-based rendering (IBR) methods tabulate the 4D radiance field, and can thus account for all effects. However, for highly reflective surfaces, it is not clear whether the required sampling density of views (for view-based IBR) or of emitted radiance per surface point (for surface-based IBR) is practical because of the amount of storage that would be entailed. We propose parameterizing environment maps instead of texture maps to support plausible movement away from and between the pre-rendered viewpoint samples while maintaining local reflections.

Our parameterized environment map (PEM) representation is layered, parameterized by viewpoint, and inferred to match ray-traced imagery. We compute environment map texture coordinates by intersecting rays with simple geometry that approximates the actual geometry in the environment. This is in contrast to PTMs where the texture coordinates are static. Layering allows different environment map geometries to be used to better approximate each layer, and also supports parallax between layers. Parameterizing environment maps by viewpoint has two benefits. First, we can capture view-dependent shading in the environment. Second, we can account for geometric error caused by approximating the environment with simpler geometry. We compute environment maps by inferring them to match a ray traced image rather than by rendering from the object center. Doing this at each viewpoint, we obtain a good match that compensates for geometric error.

PEMs provide a faithful approximation to ray-traced images at pre-rendered viewpoint samples and the ability to plausibly move away from those samples using real-time graphics hardware. Specifically, we have demonstrated results for highly specular mirror-like local reflections and showed that the same method can be used for glossy objects. Furthermore, this representation is easily supported in present-day graphics hardware, and has the desirable property of requiring only spatially coherent memory accesses.

Hybrid Rendering

To handle refractive objects away from the pre-rendered views, we developed hybrid rendering. Hybrid rendering combines ray tracing, which simulates complicated ray bouncing off local geometry, with environment maps which capture the more distant geometry. This exploits the hardware's ability to access and resample texture maps to reduce the number of ray casts and consider them in a memory-coherent order. By inferring layered EMs parameterized by viewpoint, we preserve view-dependent shading and parallax effects in the environment without performing unaccelerated ray casts through its complicated geometry. To limit local ray-tracing computation, we use a greedy ray path shading model that prunes the binary ray tree generated by refractive objects to form just two ray paths. We also restrict ray queries to triangle vertices, but perform adaptive tessellation to shoot additional rays where neighboring ray paths differ sufficiently. With these techniques, we obtain a realistic simulation of highly specular reflective and refractive objects. As for highly reflective objects, we expect image-based rendering methods to be impractical for highly refractive objects because of the amount of storage that would be entailed. Another benefit of hybrid rendering is that self-reflections are handled by ray tracing the local geometry rather than representing it as a parameterized environment map, and so good results are achieved with as much as ten times sparser sampling of environment maps.

Hybrid rendering was 25-84 times faster than a uniformly-sampled ray tracing. Both used identical ray casting code and the greedy ray path shading model. Nevertheless, hybrid rendering performance falls short of real-time, with results demonstrated at 5-20 seconds per frame. We believe significant opportunity remains both to optimize the software and parameters, and to tradeoff greater approximation error for higher speed. We note that more complicated environmental geometry will increase the benefit of our use of approximating shells. To speed up ray tracing, it may be advantageous to exploit spatial and temporal coherence, possibly combined with the use of higher-order surfaces rather than memory-inefficient polygonal tessellations. In any case, we believe that future improvement to CPU speeds and especially support for ray tracing in graphics hardware will make this approach ideal for real-time rendering of realistic shiny and glass objects.

With hybrid rendering, we have introduced the idea of combining ray-tracing with traditional graphics hardware. This is typically not done because the cost of ray-tracing can be high and unpredictable from one screen pixel to the next. The major benefit of this work is to make the cost of ray tracing *low* and *predictable* without sacrificing quality. Lower cost, but probably not higher predictability, results from our adaptive ray tracing algorithm. Two of our other techniques enhance both. We avoid large variations in the ray tree of refractive objects from one pixel to the next by substituting two ray paths. We also substitute a fixed set of simple shells for arbitrarily complex environmental geometry.

Future Implications

When graphics hardware improves, perhaps by moving away from traditional Z-buffer rendering toward support for ray-tracing, it is interesting to ask what contributions in our work will likely remain important. The texture mapping and programmable shading parts of the existing graphics pipeline are likely to remain important primitives in any future graphics hardware. We believe that texture mapping is fundamental because it efficiently encodes signals on surfaces as rectangular images, and allows different sampling rates for different signals with a common (u, v) parameterization. In addition, even with ray-tracing hardware, there will be a need to reduce complexity so that the rendering can be done in real-time. Taking advantage of texture mapping, environment mapping, and programmable shading capabilities, as we have done in our work, is likely to be instrumental for improving memory coherence and managing complexity on future graphics hardware.

Choice of Texture Parameterizations

In our work, we have chosen to exploit two specific kinds of texture parameterizations: viewpoint-independent (u, v) coordinates per vertex on each mesh, and view-dependent (u, v) coordinates computed by intersecting outgoing rays with simple geometric impostors. While these are certainly not the only parameterizations that one can imagine, we believe our choices of parameterizations have desirable compression and interactive rendering characteristics. For example, one well-known parameterization that we did not utilize is a view-dependent texture parameterization where view-based images are projectively mapped onto object surfaces. One advantage of projective textures over traditional ones is that they reduce the number of image sampling passes; projective textures require just one sampling pass for projection, whereas traditional textures involve a sampling pass for inverse rendering and another for texture filtering. Nevertheless, we believe our "intrinsic" texture parameterizations (i.e., viewpoint-independent (u,v) coordinates per vertex) are superior to view-based ones for capturing the view-independent lighting in a single texture map rather than a collection of views to obtain better compression. Additionally, disocclusions are handled without encoding which polygons are visible in which views or gathering polygons corresponding to different views in separate passes. For reflective and refractive objects, it is advantageous to use environment map parameterizations with simple geometric impostors approximating actual geometry in the environment. Predicting how reflections and refractions move as the view changes enables plausible movement away from the pre-rendered views.

Summary

In summary, we provide photorealistic rendering of multi-dimensional parameterized image spaces. A distinguishing characteristic of our work is that we generalize to image spaces with arbitrary parameters, not just radiance fields parameterized by viewpoint as in image-based rendering, or animations through 1D time as in movies. We compute texture maps by inverse rendering from offline ray-traced imagery. The parameterized textures are compressed exploiting multi-dimensional coherence, and can be quickly decoded for real-time playback on graphics hardware. With parameterized environment maps and hybrid rendering, we allow for plausible movement away from the original samples for highly specular reflective and refractive objects that would be impractical with light field based methods.

Our overall approach attempts to bridge the gap between the higher quality achievable offline and what can be done in real-time. It is reasonable to assume that there will always be a difference in quality between these two modes of rendering. We therefore expect that inverse fitting to primitives supported in real-time graphics hardware will remain an important problem.

7.2 Recommendations for Graphics Hardware

One impediment in our system to real-time playback is the meager support for texture decoding in graphics hardware. This bottleneck can be overcome by absorbing some of the decoding functionality into the hardware. Indeed, we expect the ability to load highly compressed textures directly to hardware in the near future.

If we assume that textures are stored in uncompressed form in system memory (i.e. no texture decompression is needed), downloading textures into hardware memory becomes the performance bottleneck. Two ways to alleviate this bottleneck are higher system bus bandwidth and transfer of compressed textures across the bus rather than raw images. A further enhancement would be to load compressed parameter-dependent texture block pyramids directly to hardware. By exploiting coherence between texture images in a single block, the bus bandwidth requirements would be reduced. Support for parameter-dependent texture blocks in the application programming interface (API) would make this performance optimization possible.

When splitting an object's lighting layers into specular and diffuse terms, it is necessary to compress gamma corrected textures for the individual terms. This is because gamma correction of the final image magnifies compression errors in the dark regions. One consequence of gamma correcting the individual textures is that they must be inverse gamma corrected prior to summing the two terms in the graphics hardware, where all image operations must be performed in linear space. We note that the inverse gamma function employed, as well as gamma correction at higher precision than the 8-bit framebuffer result, is a useful companion to hardware decompression.

Factoring surface reflectance from incident radiance in our parameterized environment map representation is problematic on current hardware with fixed point 8-bit texture arithmetic; it is difficult to fit the dynamic range needed by the incident specular layer [Deb98b]. We clip samples that are too bright, sometimes resulting in artificially dimmed highlights. Solving this problem will require more dynamic range in texture processing, perhaps using a floating point representation.

Finally, we would like hardware acceleration of ray-tracing for local models, so that we can perform hybrid rendering with refractive objects in real-time. At some point, we expect the current graphics architecture will be augmented with some form of ray tracing capability. In that case, our ideas of hybrid rendering and fitting accurate impostors may prove very useful to make ray tracing more local and exploit texture-mapping.

7.3 Future Work

The traditional graphics pipeline is becoming increasingly programmable [Lin01]. One area for future work is to explore the use of more sophisticated models for hardware rendering to more efficiently encode parameterized image spaces. For example, we can imagine simulating realistic lighting by inverse fitting to analytical models for area light sources that can be run on programmable hardware. We can also imagine evaluating higher-order functions on programmable hardware to tradeoff storage for computation. For example, one can compute diffuse shading effects by evaluating quadratic polynomials at each pixel [Mal01, Ram01a]. We believe that our simple sum of diffuse and specular texture maps is but a first step toward more predictive graphics models supported by hardware to aid compression. In addition, the discipline of measuring compression ratios vs. error for encodings of photorealistic imagery is perhaps a useful benchmark for proposed hardware enhancements.

Extending our work to deforming geometry should be possible using parameter-dependent geometry compression [Len99]. Another extension is to match photorealistic camera models (e.g., imagery with depth of field effects) in addition to photorealistic shading. This may be possible with accumulation-buffer methods [Hae90] or with hardware post-processing on separately rendered sprites [Sny98]. Use of perceptual metrics to guide compression and storage allocation is another important extension [Lub95]. Further work is also required to automatically generate contiguous, sampling-efficient texture parameterizations over arbitrary meshes using a minimum of maps [San01].

Another area of future work is to study the benefits of hybrid rendering on compression. We expect that PEMs should better capture the coherence in image spaces compared with parameterized texture maps that are statically mapped on objects. The dynamic computation of texture coordinates by intersecting outgoing rays with the simple imposter geometry can be considered to be a form of motion prediction. This is yet another example of taking advantage of more information about a synthetic scene to achieve greater compression.

In our hybrid rendering approach, we presently match impostors to a two-term greedy ray path shading model. Ideally, we would like to fit impostors to match the full binary tree ray-tracing rather than our ray path approximation to achieve greater realism.

During preprocessing, we currently ray-trace each image in the multi-dimensional space independently. This is very expensive and the main reason we are limited to just a few dimensions. We think there is opportunity to better exploit coherence in the image space to make the ray-tracing more efficient, using ideas similar to [Hal98]. Note that even if we have ray-tracing that exploits coherence in the pre-processing, it will clearly be impractical to store the images in uncompressed form and then do block-based compression, since even this processing is exponential in dimension.

Somehow, the ray tracing results will have to be already represented in compressed form across blocks of the parameter space.

Finally, assuming we can eliminate the ray-tracing bottleneck, then we can begin to consider spaces with higher dimensions. We are interested in measuring storage required as the dimension of the parameterized space grows and hypothesize that such growth is quite small in many useful cases because of greater coherence. We also believe that pre-processed encoding of parameterized image spaces with higher dimensions will make our approach applicable to a wider range of graphics applications.

Bibliography

[Ade91] Adelson, E., and J. Bergen, "The Plenoptic Function and the Elements of Early Vision," In *Computational Models of Visual Processing*, MIT Press, Cambridge, MA, pages 3-20, 1991.

[Agr95] Agrawala, M., A. Beers, and N. Chaddha, "Model-based Motion Estimation for Synthetic Animations," In *Proceedings of the ACM International Multimedia Conference '95*, pages 477-488, November 1995.

[Agr00] Agrawala, M., R. Ramamoorthi, A. Heirich, and L. Moll, "Efficient Image-Based Methods for Rendering Soft Shadows," In *SIGGRAPH '00*, pages 375-384, July 2000.

[Ake93] Akeley, K., "RealityEngine Graphics," In *SIGGRAPH '93*, pages 109-116, August 1993.

[Arv86] Arvo, J., "Backwards Ray Tracing," In *Developments in Ray Tracing*, SIGGRAPH '86 Course Notes, volume 12, pages 259-263, August 1986.

[Bas99] Bastos, R., K. Hoff, W. Wynn, and A. Lastra. "Increased Photorealism for Interactive Architectural Walkthroughs," In *Proceedings of the Symposium on Interactive 3D Graphics '99*, pages 183-190, April 1999.

[Bee96] Beers, A. C., M. Agrawala, and N. Chaddha, "Rendering from Compressed Textures," In *SIGGRAPH '96*, pages 373-378, August 1996.

[Bli76] Blinn, J. F., and M. E. Newell, "Texture and Reflection in Computer Generated Images," In *Communications of the ACM*, volume 19, number 10, pages 542-547, October 1976.

[Bli77] Blinn, J. F., "Models of light reflection for computer synthesized pictures," In *SIGGRAPH '77*, pages 192-198, July 1977.

[Bou70] Bouknight, W. J., "A procedure for generation of three-dimensional half-toned computer graphics presentations," In *Communications of the ACM*, volume 13, number 9, pages 527-536, September 1970.

[Bue01] Buehler, C., M. Bosse, L. McMillan, S. Gortler, and M. Cohen, "Unstructured Lumigraph Rendering," In *SIGGRAPH '01*, pages 425-432, August 2001.

[Bur83] Burt, P., and E. Adelson, "The Laplacian Pyramid as a Compact Image Code," In *IEEE Transactions on Communications*, volume Com-31, number 4, pages 532-540, April 1983.

[Cab99] Cabral, B., M. Olano, and N. Philip, "Reflection Space Image Based Rendering," In *SIGGRAPH '99*, pages 165-170, August 1999.

[Cha99] Chang, C., G. Bishop, and A. Lastra, "LDI Tree: A Hierarchical Representation for Image-Based Rendering," In *SIGGRAPH '99*, pages 291-298, August 1999.

[Che91] Chen, E. S., H. E. Rushmeier, G. Miller, and D. Turner, "A Progressive Multi-Pass Method for Global Illumination," In *SIGGRAPH '91*, pages 164-174, August 1991.

[Che93] Chen, S.E., and L. Williams, "View Interpolation for Image Synthesis," In *SIGGRAPH '93*, pages 279-288, August 1993.

[Che95] Chen, S.E., "QuickTime VR - An Image-Based Approach to Virtual Environment Navigation," In *SIGGRAPH '95*, pages 29-38, August 1995.

[Chu00] Chuang, Y., D. Zongker, J. Hindorff, B. Curless, D. Salesin, and R. Szeliski, "Environment Matting Extensions: Towards Higher Accuracy and Real-Time Capture," In *SIGGRAPH '00*, pages 121-130, July 2000.

[Coh93] Cohen, M. F., and J. R. Wallace, *Radiosity and Realistic Image Synthesis*, Academic Press, August 1993.

[Coh99] Cohen-Or, D., Y. Mann, and S. Fleishman, "Deep Compression for Streaming Texture Intensive Animations," In *SIGGRAPH '99*, pages 261-265, August 1999.

[Coo82] Cook, R. L., and K. E. Torrance, "A reflectance model for computer graphics," In *SIGGRAPH '81*, volume 15, number 3, pages 307-316, August 1981.

[Coo84a] Cook, R. L., T. Porter, and L. Carpenter, "Distributed ray tracing," In *SIGGRAPH '84*, pages 137-145, July 1984.

[Coo84b] Cook, R. L., "Shade Trees," In *SIGGRAPH '84*, pages 223-231, July 1984.

[Deb96] Debevec, P., C. Taylor, and J. Malik, "Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach," In *SIGGRAPH '96*, pages 11-20, August 1996.

[Deb98a] Debevec, P., Y. Yu, and G. Borshukov, “Efficient View-Dependent Image-Based Rendering with Projective Texture Maps,” In *9th Eurographics Workshop on Rendering*, pages 105-116, June 1998.

[Deb98b] Debevec, P., “Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography,” In *SIGGRAPH ’98*, pages 189-198, July 1998.

[Dief96] Diefenbach, P. J., *Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering*, Ph.D. Dissertation, University of Pennsylvania, 1996.

[Dor99] Dorsey, J., A. Edelman, H. W. Jensen, J. Legakis, and H. K. Pederson, “Modeling and rendering weathered stone,” In *SIGGRAPH ’99*, pages 225-234, August 1999.

[DX8] Microsoft DirectX 8.0, http://www.microsoft.com/directx/.

[Eng96] Engl, E., M. Hanke, and A. Neubauer, *Regularization of Inverse Problems*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[Fol90] Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.

[Ger92] Gersho, A., and R. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic, Boston, 1992.

[Gla89] Glassner, A. S., *An Introduction to Ray Tracing*, Academic Press, August 1989.

[Gle98] Gleicher, M., “Retargeting Motion to New Characters,” In *SIGGRAPH ’98*, pages 33-42, July 1998.

[Gor84] Goral, C. M., K. E. Torrance, D. P. Greenberg, and B. Battaile, “Modelling the interaction of light between diffuse surfaces,” In *SIGGRAPH ’84*, pages 212-222, July 1984.

[Gor96] Gortler, S., R. Grzeszczuk, R. Szeliski, and M. Cohen, “The Lumigraph,” In *SIGGRAPH ’96*, pages 43-54, August 1996.

[Gou71] Gouraud, H., “Computer display of curved surfaces,” Technical report, Department of Computer Science, University of Utah, Salt Lake City, Utah, 1971.

[Gre86] Greene, N., “Environment Mapping and Other Applications of World Projections,” In *IEEE Computer Graphics and Applications*, volume 6, number 11, pages 21-29, November 1986.

[Gue93] Guenter, B., H. Yun, and R. Mersereau, “Motion Compensated Compression of Computer Animation Frames,” In *SIGGRAPH ’93*, pages 297-304, August 1993.

[Hae90] Haeberli, P., and K. Akeley, “The Accumulation Buffer: Hardware Support for High-Quality Rendering,” In *SIGGRAPH ’90*, pages 309-318, August 1990.

[Hae93] Haeberli, P., and M. Segal, “Texture Mapping as a Fundamental Drawing Primitive,” In *4th Eurographics Workshop on Rendering*, pages 259-266, 1993.

[Hak00] Hakura, Z. S., J. E. Lengyel, and J. M. Snyder, “Parameterized Animation Compression,” In *11th Eurographics Workshop on Rendering*, pages 101-112, June 2000.

[Hak01a] Hakura, Z. S., J. M. Snyder, and J. E. Lengyel, “Parameterized Environment Maps,” In *Proceedings of the Symposium on Interactive 3D Graphics ’01*, pages 203-208, March 2001.

[Hak01b] Hakura, Z. S., and J. M. Snyder, “Realistic Reflections and Refractions on Graphics Hardware with Hybrid Rendering and Layered Environment Maps,” In *12th Eurographics Workshop on Rendering*, June 2001.

[Hal98] Halle, M., “Multiple Viewpoint Rendering,” In *SIGGRAPH ’98*, pages 243-254, August 1998.

[Han90] Hanrahan, P. and J. Lawson, “A Language for Shading and Lighting Calculations,” In *SIGGRAPH ’90*, pages 289-298, August 1990.

[Han93] Hanrahan, P. and W. Krueger, “Reflection from layered surfaces due to subsurface scattering,” In *SIGGRAPH ’93*, pages 165-174, August 1993.

[Han97] Hanrahan, P., *The Visual Computer*, Invited State-of-the-Field Talk, Supercomputing ’97, November 1997.

[Hec87] Hecht, E., *Optics*, Second Edition, Addison-Wesley, 1987.

[Hei99a] Heidrich, W., H. Lensch, and H. P. Seidel, “Light Field Techniques for Reflections and Refractions,” In *10th Eurographics Workshop on Rendering*, pages 195-204, June 1999.

[Hei99b] Heidrich, W. and H. P. Seidel., “Realistic, Hardware-Accelerated Shading and Lighting,” In *SIGGRAPH ’99*, pages 171-178, August 1999.

[Hei00] Heidrich, W., K. Daubert, J. Kantz, and H.-P. Seidel, “Illuminating Micro Geometry Based on Precomputed Visibility,” In *SIGGRAPH ’00*, pages 455-464, July 2000.

[Imm86] Immel, D. S., M. Cohen and D. P. Greenberg, “A radiosity method for non-diffuse environments,” In *SIGGRAPH ’86*, pages 133-142, August 1986.

[Jen96] Jensen, H. W., *The Photon Map in Global Illumination*, Ph.D. Dissertation, Technical University of Denmark, September 1996.

[Jen01] Jensen, H. W., S. R. Marschner, M. Levoy, and P. Hanrahan, “A Practical Model for Subsurface Light Transport,” In *SIGGRAPH ’01*, pages 511-518, August 2001.

[Kaj85] Kajiya, J. T., “Anisotropic reflection models,” In *SIGGRAPH ’85*, volume 19, pages 15-21, July 1985.

[Kaj86] Kajiya, J. T., “The rendering equation,” In *SIGGRAPH ’86*, pages 143-150, August 1986.

[Kaj89] Kajiya, J. T., and T. L. Kay, “Rendering Fur with Three Dimensional Textures,” In *SIGGRAPH ’89*, pages 271-280, July 1989.

[Kau99] Kautz, J. and M. D. McCool, “Interactive Rendering with Arbitrary BRDFs using Separable Approximations,” In *10th Eurographics Wokshop on Rendering*, pages 255-268, June 1999.

[Kay79] Kay, D., and D. Greenberg, “Transparency for Computer Synthesized Images,” In *SIGGRAPH ’79*, volume 13, number 2, pages 158-164, August 1979.

[Lal99] Lalonde, P. and A. Fournier, “Interactive Rendering of Wavelet Projected Light Fields,” In *Graphics Interface ’99*, pages 107-114, June 1999.

[Leg91] Legall, D., “MPEG: A video compression standard for multimedia applications,” In *Communications of the ACM*, volume 34, number 4, pages 46-58, April 1991.

[Len99] Lengyel, J., “Compression of Time-Dependent Geometry,” In *Proceeding so the Symposium on Interactive 3D Graphics ’99*, pages 89-96, April 1999.

[Len00] Lengyel, J., “Real-Time Hair,” In *11th Eurographics Workshop on Rendering*, pages 243-256, June 2000.

[Len01] Lengyel, J., “Real-Time Fur over Arbitrary Surfaces,” In *Proceedings of the Symposium on Interactive 3D Graphics ’01*, pages 227-232, March 2001.

[Lev95] Levoy, M., “Polygon-Assisted JPEG and MPEG compression of Synthetic Images,” In *SIGGRAPH ’95*, pages 21-28, August 1995.

[Lev96] Levoy, M., and P. Hanrahan, “Light Field Rendering,” In *SIGGRAPH ’96*, pages 31-41, August 1996.

[Lin80] Linde, Y., A. Buzo, and R. M. Gray, “An algorithm for Vector Quantizer Design,” In *IEEE Transactions on Communication*, volume COM-28, number 1, pages 84-95, January 1980.

[Lin01] Lindholm, E., M. J. Kilgard, and H. Moreton, “A User-Programmable Vertex Engine,” In *SIGGRAPH ’01*, pages 149-158, August 2001.

[Lis98] Lischinski, D., and A. Rappoport, “Image-Based Rendering for Non-Diffuse Synthetic Scenes,” In *9th Eurographics Workshop on Rendering*, pages 301-314, June 1998.

[Lub95] Lubin, J., “A Visual Discrimination Model for Imaging System Design and Evaluation,” In E. Peli, editor, *Vision Models for Target Detection and Recognition*, World Scientific, pages 245-283, 1995.

[Lue94] Luettgen, M., W. Karl, and A Willsky, “Efficient Multiscale Regularization with Applications to the Computation of Optical Flow,” In *IEEE Transactions on Image Processing*, volume 3, number 1, pages 41-63, January 1994.

[Mag00] Magnor, M. A., *Geometry-Adaptive Multi-View Coding Techniques for Image-Based Rendering*, Ph.D. Dissertation, University of Erlangen-Nuremberg, Germany, November 2000.

[Mai93] Maillot, J., H. Yahia, A. Verroust, “Interactive Texture Mapping,” In *SIGGRAPH ’93*, pages 27-34, August 1993.

[Mal01] Malzbender, T., D. Gelb, and H. Wolters, “Polynomial Texture Maps,” In *SIGGRAPH ’01*, pages 519-528, August 2001.

[Mar98] Marschner, S. R., *Inverse Rendering for Computer Graphics*, Ph.D. Dissertation, Cornell University, August 1998.

[McC01] McCool, M., J. Ang, and A. Ahmad, “Homomorphic Factorization of BRDFs for High-Performance Rendering,” In *SIGGRAPH ’01*, pages 171-178, August 2001.

[McM95] McMillan, L., and G. Bishop, “Plenoptic Modeling,” In *SIGGRAPH ’95*, pages 39-46, August 1995.

[MIC99] Microsoft MPEG-4 Visual Codec FDIS 1.02, ISO/IEC 14496-5 FDIS1, August 1999.

[Mil84] Miller, G. S., and C. R. Hoffman, “Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments,” In *SIGGRAPH ’84: Advanced Computer Graphics Animation Seminar Notes*, July 1984.

[Mil88] Miller, G. S. P., “From Wire-Frames to Furry Animals,” In *Graphics Interface ’88*, pages 138-145, June 1988.

[Mil98a] Miller, G., S. Rubin, and D. Poncelen, “Lazy Decompression of Surface Light Fields for Pre-computed Global Illumination,” In *9th Eurographics Workshop on Rendering*, pages 281-292, June 1998.

[Mil98b] Miller, G., M. Halstead, M. Clifton, “On-the-Fly Texture Computation for Real-Time Surface Shading,” In *IEEE Computer Graphics and Applications*, volume 18, number 2, pages 44-58, March/April 1998.

[Nei93] Neider, J., T. Davis, and M. Woo, *OpenGL Programming Guide*, Addison-Wesley, 1993.

[Nim94] Nimeroff, J., E. Simoncelli, and J. Dorsey, “Efficient Re-rendering of Naturally Illuminated Environments,” In *5th Eurographics Workshop on Rendering*, pages 359-374, June 1994.

[Nis99] Nishino, K., Y. Sato, and K. Ikeuchi, “Eigen-Texture Method: Appearance Compression based on 3D Model,” In *Proceedings of 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 618-624, June 1999.

[NV] NVIDIA Corporation web site, http://www.nvidia.com/.

[Ofek98] Ofek, E. and A. Rappoport, “Interactive Reflections on Curved Objects,” In *SIGGRAPH ’98*, pages 333-342, July 1998.

[Pen92] Pennebaker, W., and J. Mitchell, *JPEG Still Image Compression Standard*, New York, Van Nostrand Reihold, 1992.

[Per85] Perlin, K., “An Image Synthesizer,” In *SIGGRAPH ’85*, pages 287-296, July 1985.

[Pha97] Pharr, M., C. Kolb, R. Gershbein, and P. Hanrahan, “Rendering Complex Scenes with Memory-Coherence Ray Tracing,” In *SIGGRAPH ’97*, pages 101-108, August 1997.

[Pha00] Pharr, M., and P. Hanrahan, “Monte Carlo evaluation of non-linear scattering equations for subsurface reflection,” In *SIGGRAPH ’00*, pages 75-84, July 2000.

[Pho75] Phong, B. T., “Illumination for computer generated pictures,” In *Communications of the ACM*, volume 18, number 6, pages 311-317, June 1975.

[Pop99] Popović, Z., and A. Witkin, “Physically Based Motion Transformation,” In *SIGGRAPH ’99*, pages 11-20, August 1999.

[Ram01a] Ramamoorthi, R., and P. Hanrahan, “An Efficient Representation for Irradiance Environment Maps,” In *SIGGRAPH ’01*, pages 497-500, August 2001.

[Ram01b] Ramamoorthi, R., and P. Hanrahan, “A Signal-Processing Framework for Inverse Rendering,” In *SIGGRAPH ’01*, pages 117-128, August 2001.

[Ren89] *The RenderMan Interface*, PIXAR, December 1989.

[Ros98] Rose, C., M. Cohen, and B. Bodenheimer, “Verbs and Adverbs: Multidimensional Motion Interpolation,” In *IEEE Computer Graphics and Applications*, volume 18, number 5, pages 32-40, September 1998.

[Sai96] Said, A., and W. Pearlman, “A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees,” In *IEEE Transactions on Circuits and Systems for Video Technology*, volume 6, number 3, pages 243-250, June 1996.

[San01] Sander, P. V., S. J. Gortler, J. Snyder, and H. Hoppe, “Texture-Mapping Progressive Meshes,” In *SIGGRAPH ’01*, pages 409-416, August 2001.

[Sat97] Sato, Y., M. Wheeler, and K. Ikeuchi, “Object Shape and Reflectance Modeling from Observation,” In *SIGGRAPH ’97*, pages 379-387, August 1997.

[Sch93] Schlick, C., “A Customizable Reflectance Model for Everyday Rendering,” In *4th Eurographics Workshop on Rendering*, pages 73-83, June 1993.

[Seg92] Segal, M., and K. Akeley, *The OpenGL Graphics System: A Specification*, 1992.

[Sha98] Shade, J., S. Gortler, L. He, and R. Szeliski, “Layered Depth Images,” In *SIGGRAPH ’98*, pages 75-82, August 1998.

[Shi90] Shirley, P., “A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes,” In *Graphics Interface ’90*, pages 205-212, 1990.

[Shi92] Shirley, and Wang, “Distribution Ray Tracing: Theory and Practice,” In *3rd Eurographics Workshop on Rendering*, pages 33-43, 1992.

[Shi95] Shirley, P., B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg, “Global Illumination via Density Estimation,” In *6th Eurographics Workshop on Rendering*, pages 219-230, 1995.

[Shi96] Shirley, Wang and Zimmerman, “Monte Carlo Methods for Direct Lighting Calculations,” In *ACM Transactions on Graphics*, volume 15, number 1, pages 1-36, January 1996.

[Shu99] Shum, H.-Y., and L.-W. He, “Rendering with Concentric Mosaics,” In *SIGGRAPH ’99*, pages 299-306, August 1999.

[Sil89] Sillion, F. X. and C. Puech, “A general two-pass method integrating specular and diffuse reflection,” In *SIGGRAPH ’89*, pages 335-344, August 1989.

[Sny98] Snyder, J., and J. Lengyel, “Visibility Sorting and Compositing without Splitting for Image Layer Decompositions,” In *SIGGRAPH ’98*, pages 219-230, August 1998.

[Sta95] Stam, J., “Multiple scattering as a diffusion process,” In *6th Eurographics Workshop on Rendering*, pages 41-50, June 1995.

[Sta99] Stamminger, M., A. Scheel, et al., “Efficient Glossy Global Illumination with Interactive Viewing,” In *Graphics Interface ’99*, pages 50-57, June 1999.

[Stu97] Strzlinger, W. and R. Bastos. “Interactive Rendering of Globally Illuminated Glossy Scenes,” In *8th Eurographics Workshop on Rendering*, pages 93-102, June 1997.

[Teo97] Teo, P., E. Simoncelli, and D. Heeger, “Efficient Linear Re-rendering for Interactive Lighting Design,” Stanford Computer Science Department Technical Report STAN-CS-TN-97-60, October 1997.

[Ter86] Terzopoulos, D., “Regularization of Inverse Visual Problems Involving Discontinuities,” In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 8, number 4, pages 413-424, July 1986.

[Ton00] Tong, X., and R. M. Gray, “Coding of multi-view images for immersive viewing,” In *Proceedings IEEE International Conference on Acoustic, Speech, and Signal Processing*, volume 4, pages 1879-1882, June 2000.

[Ton01] Tong, X., and R. M. Gray, “Interactive View Synthesis from Compressed Light Fields,” In *Proceeding IEEE International Conference on Image Processing*, October 2001.

[Ude99] Udeshi, T. and C. Hansen. “Towards interactive, photorealistic rendering of indoor scenes: A hybrid approach,” In *10th Eurographics Workshop on Rendering*, pages 71-84, June 1999.

[Vea97] Veach, E., *Robust Monte Carlo Methods for Light Transport Simulation*, Ph.D. Dissertation, Stanford University, December 1997.

[Ver84] Verbeck, C. P. and D. P. Greenberg, “A Comprehensive Light-Source Description for Computer Graphics,” In *IEEE Transactions on Computer Graphics and Applications,* pages 253-360, July 1984.

[Voo94] Voorhies, D., and J. Foran, “Reflection Vector Shading Hardware,” In *SIGGRAPH ’94*, pages 163-166, July 1994.

[Wal87] Wallace, J. R., M. F. Cohen, and D. P. Greenberg, “A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods,” In *SIGGRAPH ’87*, pages 311-320, July 1987.

[Wal94] Wallach, D. S., S. Kunapalli, and M. F. Cohen, “Accelerated MPEG Compression of Dynamic Polygonal Scenes,” In *SIGGRAPH ’94*, pages 193-196, July 1994.

[Wal97] Walter, B., G. Alppay, E. Lafortune, S. Fernandez, and D. Greenberg, “Fitting Virtual Lights for Non-Diffuse Walkthroughs,” In *SIGGRAPH ’97*, pages 45-48, August 1997.

[War88] Ward, G. J, F. M. Rubinstein, and R. D. Clear, “A Ray Tracing Solution for Diffuse Interreflection,” In *SIGGRAPH ’88*, pages 85-92, August 1988.

[Whi80] Whitted, T., “An improved illumination model for shaded display,” In *Communications of the ACM*, volume 23, number 6, pages 343-349, June 1980.

[Wil83] Williams, L., “Pyramidal Parametrics,” In *SIGGRAPH ’83*, pages 1-11, July 1983.

[Won97] Wong, T. T., P. A. Heng, S. H. Or, and W. Y. Ng, “Image-based Rendering with Controllable Illumination,” In *8th Eurographics Rendering Workshop*, pages 13-22, June 1997.

[Woo00] Wood, D. N., Azuma, D. I., Aldinger, K. et al., “Surface Light Fields for 3D Photography,” In *SIGGRAPH ’00*, pages 287-296, July 2000.

[Wu00] Wu, Y., L. Luo, J. Li and Y.-Q. Zhang, “Rendering of 3D wavelet compressed concentric mosaic scenery with progressive inverse wavelet synthesis (PIWS),” In *Proceedings SPIE Visual Communications and Image Processing*, volume 1, pages 31-42, June 2000.

[Xio96] Xiong, Z., O. Guleryuz, and M. Orchard, “A DCT-based Image Coder,” In *IEEE Signal Processing Letters*, November 1996.

[Yu99] Yu, Y., P. Debevec, J. Malik, and T. Hawkins, “Inverse Global Illumination: Recovering Reflectance Models of Real Scenes From Photographs,” In *SIGGRAPH ’99*, pages 215-224, August 1999.

[Zha00] Zhang, C., and J. Li, “Compression and rendering of concentric mosaics with reference block codec (RBC),” In *Proceedings SPIE Visual Communications and Image Processing*, volume 1, pages 43-54, June 2000.

[Zon99] Zongker, D., D. Werner, B. Curless, and D. Salesin, “Environment Matting and Composition,” In *SIGGRAPH ’99*, pages 205-214, August 1999.