PIPELINE RENDERING: INTERACTION AND REALISM THROUGH HARDWARE-BASED MULTI-PASS RENDERING

PAUL JOSEPH DIFEENBACH

A DISSERTATION
in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.
1996

______________________________
Norman I. Badler
Supervisor of Dissertation

______________________________
Peter Buneman
Graduate Group Chairperson
Acknowledgments

_Lazlo’s Chinese Relativity Axiom:_
No matter how great your triumphs or how tragic your defeats,
approximately one billion Chinese couldn’t care less.

There are many people who contributed to this thesis both in direct support as well as
through personal support, and I am greatly indebted to them all. Foremost, I would
like to thank my advisor, Dr. Norman Badler, whose confidence started me along the
doctoral path, whose support enabled my research, and whose expectations helped
push the final result. I am sincerely thankful to my other committee members,
namely Dimitri Metaxas, Insup Lee, Julie Dorsey, and Greg Ward, who provided
valuable direction in turning a disjoint proposal into a cohesive, directed thesis. I am
particularly indebted to Greg Ward, whose expertise and guidance helped strengthen
the rendering aspects of this work, and whose bathroom model and images provided
an invaluable testbed.

I am also immensely thankful for the support, friendship, and help of the students
and staff of the Center for Human Modeling and Simulation. They have always been
willing to readily share their knowledge and expertise (as well as their machines)
when needed. In particular, I am grateful to Jeff Nimeroff for his help with ren-
dering jargon and his bibtex-like knowledge, to Doug DeCarlo for his help with SGI
architectural quirks and for our discussions on shadow volumes, and to John Granieri
for providing the impetus for the specular image work. I would also like to thank
Min-Zhi Shao for his help with his radiosity program to garner the indirect solution.

Additionally, I am appreciative to those people outside the university who pro-
vided assistance with my research and implementation. This includes Jim Arvo, who
was most generous in providing his images and the intricate butterfly environments, and Paul Heckbert who provided additional shadowing references.

On a personal note, I am extremely lucky to have many friends and family which provided innumerable support through these many years, and managed to keep me (mostly) sane throughout. To my best friend, Dr. Jana Kosecka, who began and ended (3 days early) this endeavor with me, you have been my “dub strom” as well, and I will miss you more than you know. Without you, I would never have finished. I am also fortunate to have had a couple of other friends at Penn which have also gone beyond what anyone could hope for in always looking out for me and who have meant so much to me. This includes former lab member Leanne Hwang, one of the sweetest people I know, and who, with Min, kept me fed despite giving me grey hairs; and includes Diane Chi who literally went out of her way to be a tremendous friend and who, with Rox, proved to be always full of “surprises”. I am also grateful for my continuing friendship with Kasturi Bagchi, Esq., who suffered through my paper deadlines and was my personal “advocate”, inspiration, and escape throughout this work. To my longtime friends including Robert, Andy, Gary, Glenn and Auts, Sarah, Jui and Jim, May, Susanne, and Steve, I am very thankful for the all-too-infrequent times we were able to get together during this work. There are many other friends not mentioned but who are never far from my mind.

Lastly, but most importantly, I am fortunate to have been raised in a loving and supportive family, which provided the foundation for everything I have achieved. I have no means to truly thank my mom, Natalie, who is the mother every child would wish for and whose unconditional love and support has always been the one constant throughout my life. I dedicate this thesis to you. I thank my dad, Joseph, for instilling in me a sense of independence and hard work, and for pushing me to do my best. I also am grateful to have my Grandma Marie, my sister Terri and brother-in-law Rob, and my amazing nephew Evan, all in my life. They, along with my other relatives, always make coming home truly “home”.

iv
Abstract
While large investments are made in sophisticated graphics hardware, most realistic rendering is still performed off-line using ray trace or radiosity systems. A coordinated use of hardware-provided bitplanes and rendering pipelines can, however, approximate ray trace quality illumination effects in a user-interactive environment, as well as provide the tools necessary for a user to declutter such a complex scene. A variety of common ray trace and radiosity illumination effects are presented using multi-pass rendering in a pipeline architecture. We provide recursive reflections through the use of secondary viewpoints, and present a method for using a homogeneous 2-D projective image mapping to extend this method for refractive transparent surfaces. This paper then introduces the Dual Z-buffer, or DZ-buffer, an evolutionary hardware extension which, along with current frame-buffer functions such as stencil planes and accumulation buffers, provides the hardware platform to render non-refractive transparent surfaces in a back-to-front or front-to-back order. We extend the traditional use of shadow volumes to provide reflected and refracted shadows as well as specular light reclassification. The shadow and lighting effects are then incorporated into our recursive viewpoint paradigm. Global direct illumination is provided through a shadow blending technique. Hardware surface illumination is fit to a physically-based BRDF to provide a better local direct model, and the framework permits incorporation of a radiosity solution for indirect illumination as well. Additionally, we incorporate material properties including transluency, light scattering, and non-uniform transmittance to provide a general framework for creating realistic renderings. The DZ-buffer also provides decluttering facilities such as transparency and clipping. This permits selective scene viewing through arbitrary view-dependent and non-planar clipping and transparency surfaces in real-time. The combination of these techniques provide for understandable, realistic scene rendering at typical rates 5-50 times that of a comparable ray trace images. In addition, the pixel-parallel nature of these methods leads to exploration of further hardware rendering engine extensions which can exploit this coherence.
Contents

Acknowledgments iii

Abstract v

1 Introduction and Motivation 1
  1.1 Problems with interactive rendering ......................... 3
  1.2 Our Approach ............................................. 6
  1.3 Overview .................................................. 10

2 Background 12
  2.1 Rendering Methods .......................................... 12
    2.1.1 Basic Recursive Ray Tracing ............................ 12
    2.1.2 Backward Ray Tracing ................................ 14
    2.1.3 Radiosity ............................................. 16
    2.1.4 Two-Pass Methods .................................... 17
    2.1.5 Beam Tracing ........................................ 17
    2.1.6 Hardware-based Rendering ............................. 19
  2.2 Definitions ................................................. 21
    2.2.1 Stencil Planes ........................................ 21
    2.2.2 Accumulation Buffer ................................. 22
    2.2.3 Alpha Blending ...................................... 23
    2.2.4 Shadow Volumes ...................................... 23
2.2.5 Light Volumes ........................................... 24
2.2.6 In-Out Refractions ................................... 24

3 Specular Surface Rendering ........................................ 25
  3.1 Reflections ............................................. 25
  3.2 Refractive Transparency .................................. 28
  3.3 Non-Refractive Transparency ................................ 34
    3.3.1 Definitions ..................................... 36
    3.3.2 Back-to-Front Transparency ......................... 37
    3.3.3 Front-to-Back Transparency ........................ 42
  3.4 Translucency ........................................... 46

4 Shadow and Light Volumes ........................................... 52
  4.1 Shadow Volumes ......................................... 52
    4.1.1 Silhouette Volume ................................ 54
    4.1.2 View in Volume .................................. 55
  4.2 Specular Shadows ........................................ 57
    4.2.1 Intersection Recursion ............................. 58
    4.2.2 Virtual Position Recursion ......................... 60
  4.3 Light Volumes ........................................... 66
    4.3.1 Single-Bounce Light Volumes ....................... 66
    4.3.2 Multiple-Bounce Light Volumes .................... 67
  4.4 Light Accumulation ..................................... 69
    4.4.1 Attenuation ...................................... 72
    4.4.2 Filtering ......................................... 73

5 Multi-pass Process .............................................. 78
  5.1 Recursion .............................................. 78
  5.2 Stenciling .............................................. 79
6 Illumination Model

6.1 Global Direct Illumination ........................................ 85
   6.1.1 Light Accumulation ......................................... 86
   6.1.2 Soft Shadows .................................................. 87
6.2 Local Direct Illumination .......................................... 90
6.3 Indirect Illumination .............................................. 101
6.4 Total Scene Illumination .......................................... 102

7 Performance

7.1 Diffuse Transport .................................................. 111
7.2 Shadows ............................................................. 111
7.3 Specular Images .................................................... 113
7.4 Scene Dynamics ..................................................... 115

8 Scene Uncluttering

8.1 Background .......................................................... 117
   8.1.1 Clipping Planes .............................................. 118
   8.1.2 CSG ............................................................. 118
8.2 Clipping Surface ................................................... 119
8.3 Transparency Surface ............................................. 120
8.4 Software Emulation ............................................... 120

9 Extensions

9.1 Limitations ......................................................... 123
9.2 Hardware Extensions ............................................... 124
9.3 Hardware Platforms ............................................... 125
   9.3.1 Object-Order Rasterization Systems ...................... 126
   9.3.2 Image-Order Rasterization Systems ...................... 129
   9.3.3 Hybrid Systems .............................................. 130
10 Conclusion

10.1 Contributions ................................................. 132
10.2 Future Work .................................................. 134
10.3 Conclusion ...................................................... 135

A Projective Maps .............................................. 137

B Refraction Approximation ................................. 139
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Frame rendering statistics for bathroom environment</td>
<td>109</td>
</tr>
<tr>
<td>7.2</td>
<td>Single image/single light rendering statistics for bathroom environment</td>
<td>110</td>
</tr>
<tr>
<td>7.3</td>
<td>Varying Shadow Settings</td>
<td>113</td>
</tr>
<tr>
<td>7.4</td>
<td>Varying Specular Settings</td>
<td>115</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Multi-pass Pipeline Rendering Process ............................................. 8
1.2 Multi-pass Pipeline Rendering Image ................................................ 9

2.1 Ray Trace Rays .................................................................................. 13
2.2 Caustic Polygons from Light Beam Tracing ......................................... 16
2.3 Virtual viewpoint for reflected beam ................................................... 19
2.4 Z-buffer based rendering pipeline ....................................................... 19
2.5 Shadow volumes ............................................................................... 20

3.1 Recursive Reflection ........................................................................... 27
3.2 Refracted Image vs. Camera Image ..................................................... 29
3.3 Undistorted and Distorted Refraction ................................................ 30
3.4 Tangent Law for paraxial rays ............................................................. 32
3.5 Uncorrected Refraction .................................................................... 32
3.6 Corrected Refraction .......................................................................... 33
3.7 Back-to-front rendered area ............................................................... 38
3.8 Back-to-front iterations .................................................................... 41
3.9 Front-to-back iterations ..................................................................... 43
3.10 Partial Specular Image ...................................................................... 46
3.11 Frosted Glass ..................................................................................... 49
3.12 Glossy Table ..................................................................................... 50

4.1 Shadow volume clipped by front clipping plane. ............................... 55
4.2 Light interaction with refractive surface .......................... 58
4.3 Refracted Caustics and Shadows ........................................ 59
4.4 Reflected light and shadows ............................................. 60
4.5 Shadow volumes generated involving $R_1$ ............................ 63
4.6 Light volume clipped by both refracting planes ...................... 68
4.7 Light volume clipped by both reflecting planes ..................... 69
4.8 VPR Light Volumes .................................................... 70
4.9 VPR Lighting and Shadows ............................................. 71
4.10 Light and view coordinate systems .................................. 75
4.11 Projected Textures ..................................................... 76
4.12 Projected Texture Light Pattern ..................................... 77

5.1 Recursive Image Rendering ........................................... 80
5.2 Recursive Stenciling ..................................................... 81

6.1 Jittered Light Shadow Volumes ...................................... 89
6.2 Jitter Approximation Shadow Volumes ............................... 89
6.3 BRDF Scattering Angles ............................................. 91
6.4 Phong BRDF Notation ................................................ 92
6.5 Specular Scattering Components ($\theta_i = \pi/6$) ...................... 93
6.6 Gaussian vs. Phong Specular Term ................................ 94
6.7 Simplified Gaussian vs. Fit Phong ................................ 95
6.8 Full Gaussian vs. Fit Phong ........................................ 96
6.9 Partial Gaussian vs. Fit Phong ..................................... 97
6.10 Full Gaussian vs. Phong vs. Fit Phong (view 1) .................. 98
6.11 Full Gaussian vs. Phong vs. Fit Phong (view 2) .................. 99
6.12 Gaussian vs. Phong Specular Highlights ........................... 100
6.13 Calculating Indirect Illumination Image ........................... 103
6.14 Gaussian vs. Phong Scene Illumination ........................... 105
7.1 Rendering Relationship ........................................... 107
7.2 Pipeline Rendering Relationship ................................. 108
7.3 Varying Shadow Settings ....................................... 112
7.4 Varying Specular Settings ...................................... 114
7.5 Varying Scene Geometry and Lighting ...................... 116

8.1 Clipping/Transparency Surface ................................. 121
Chapter 1

Introduction and Motivation

“Ok, wait a moment here. Hold on. There. Now you see the change was made in real time.”

*Frame showperson at Seybold conference*

As the power of today’s graphics workstations has increased, so too have the demands of the user. Whereas realism and interaction were previously mutually exclusive, today’s graphic workstations are providing the platform to develop applications with photo-realistic, interactive, dynamic, comprehensible environments. Unfortunately, today’s applications generally do not permit or take advantage of all of these features.

Traditional interactive computer graphics developed from the early military applications such project SAGE in the late 1950s. SAGE, for Semi-Automatic Ground Environment system, provided a CRT radar-like display of potential Soviet bomber targets and interceptors with a light-pen interface for assigning interceptors to targets. Symbols and identifiers replaced radar blips as the graphical representation.

CAD applications also began to develop around this time with systems such as APT (Automatically Programmed Tooling) allowing specification of part geometry and milling paths. Specification of this, however, was performed entirely off-line.

In 1963, one of the first presentations of interactive graphics on a CRT system was made by Sutherland at the Joint Computer Conference [Sut63]. His Sketchpad
system enabled interactive creation and manipulation of parts using display primitives such as lines, arcs, etc. This system introduced the notion of hierarchies based on a set of graphic primitives. It also introduced many interaction techniques using the keyboard and lightpen. This system provided the foundation of modern graphics packages and libraries such as PHIGS\cite{Hew84} and SGI’s GL\cite{SGI90}.

The need for computer-generated imagery for flight simulators led to the development of raster graphics systems. Raster graphics were introduced in the early 1970s with systems such as the Alto from Xerox PARC. This permitted filled, colored, solid surfaces which had been essentially unachievable on vector displays. This system provided the foundation for the modern realistic images which are common in today’s interactive applications.

Hidden surface removal in these raster systems led to a variety of sorting algorithms \cite{NNS72}\cite{War69}\cite{Sch69b}, and eventually led to the development of the Z-buffer \cite{Cat74} which resolved visibility conflicts at the pixel level. Some of these early hidden-surface removal methods\cite{App68}\cite{GN71} also introduced the notion of ray casting\cite{Rot82}, which later formed the bases of modern photo-realistic rendering techniques.

Whitted \cite{Whi80} introduced ray tracing as a means to integrate reflection, refraction, hidden surface removal, and shadows into a single model. This paradigm provided the means to build photo-realistic images one pixel at a time by casting rays from an eye-point through the pixel into the environment and tracing its path. As this method operates on a pixel-by-pixel basis, it is inherently non-interactive. Ray tracing techniques continued to expand in the 1980s and 1990s to include variations such as backward ray tracing and distribution ray tracing, as increasingly accurate physical properties such as caustics and shadow refraction were desired.

Other methods were developed to produce photo-realistic images, the most notable being radiosity introduced to computer graphics by Goral et al. \cite{GTGB84}. This method relies on energy exchange between surfaces in the environment, a calculation which also makes this method non-interactive. This calculation is performed
as a pre-computation on a static environment. Interactive view changes are possible in this environment, however the necessity of a static environment precludes any real user manipulation of the environment.

Hardware-based graphics continued to evolve, permitting sophisticated real-time features such as texture mapping, reflection mapping, transparency, and shadows. While many of these features are implemented as visual approximations instead of the physics-based ray tracing and radiosity solutions, they do provide realistic looking images at interactive rates.

The current state of computer graphics has in essence diverged into two areas: one based on off-line calculations to produce non-interactive, physically-based, photo-realistic images; the other based on hardware-implemented calculations to produce real-time, physically-approximated, pseudo-realistic images. Each of these approaches has its advantages and shortcomings.

1.1 Problems with interactive rendering

Much attention has been devoted to photo-realistic rendering techniques as ray tracing and radiosity packages have become increasingly sophisticated. These methods provide a basic foundation of visual cues and effects to produce extremely high quality and highly accurate images at a considerate cost, namely, computation time. Neither of these techniques have any widespread application in true interactive and dynamic environments, such as animation creation and virtual worlds.

Hardware-based 3-D graphics systems provide pseudo-realistic images at interactive rates through use of minute geometric detailing and other visual cues. Sophisticated graphics pipelines permit real-time texturing and shadows, in addition to a variety of basic lighting effects. These systems do not provide the sophisticated lighting effects and material properties that the photo-realistic systems do.

In order to clarify the limitations of the above described rendering approaches, each is individually addressed.
Ray Tracing

Systems based on forward ray tracing [Gla89] are non-interactive and suffer from problems inherent in the technique [WW92] such as costly intersection testing and incorrect specular illumination. In addition, only a few attempt to accurately handle indirect illumination [Kaj86]. Backward ray tracing systems [Arv86][HH84][CF87] more accurately handle caustics; but again these methods are very time-intensive and not remotely interactive. Even the fastest ray tracing systems require static geometry to achieve their results [SS89b].

Radiosity

Many so-called interactive environments such as Virtual Building systems [ARB90] [TS91] rely on precomputation of static environments to form progressive radiosity solutions. Other systems dealing with lighting effects [Dor93] rely on a series of images from a single viewpoint. All of the systems suffer from large computational overhead and unchangeable geometry. Even in incremental radiosity solutions [Che90], geometry changes require significant recomputation time. In addition, radiosity-based solutions inhibit the use of reflective and refractive surfaces. Ray trace/radiosity multi-pass or combined systems [WCG87][PSV90] enable this specularity, but only image-based systems [CW93][NDR95] permit any level of dynamic interaction although they sacrifice image resolution.

Geometric detail

In systems which provide realism through minute geometric detailing, this vast amount of data itself presents several problems. Whereas previous applications used graphics to simulate individual parts of a complex environment, current applications focus on visualization of the entire system together at maximal resolution. Where a small CAD part or a single room used to be the level of detail supported, current architectures and techniques now permit visualization of an entire airplane or
building at the same resolution of detail. Interactive rates have been maintained in systems such as walk-through packages by using complex view dependencies to limit the amount of data needed at any particular view, e.g. [TS91]. Cohesive CAD, analysis, and visualization tools used in Simulation Based Design systems also require this data management, in addition to sophisticated means of presentation and user selection of desired features. While these systems may be able to manage and present these vast amounts of visual data, visual overload will result if the user is not able to disregard unwanted sections.

Many techniques currently exist to minimize or de-clutter unwanted visual information. Two of the most frequently used are clipping surfaces and transparency. Unfortunately, today’s hardware-based graphic systems do not handle either of these in a wholly satisfactory manner. While the use of arbitrarily oriented clipping planes is common in many graphics systems, their use is limited to planar visual cuts. This presents a broad elimination of visual data, whereas a fine directed or sculptured cut is often desired such as in medical image reconstruction and visualization. Transparency is also available through the use of alpha blending, however the actual rendering is often incorrect: correct transparency requires depth-ordered drawing of the surfaces, which does not comply with the Z-buffer based sorting procedure used in almost all graphics systems. In addition, with the increased reliance on hardware texture mapping to add visual complexity, semi-transparent textures will further stress the need for a correct hardware-based rendering of non-opaque surfaces.

Graphics pipeline rendering

Advanced hardware architectures such as the SGI Reality Engine\textsuperscript{TM} have brought an added level of realism and interaction to dynamic environments through the use of sophisticated graphics pipelines and added levels of screen buffer information. These features have enabled software developers to bring previously unavailable details such as shadows and mirrors to many interactive applications, as well as allow
the user to selectively see desired details by clipping away unwanted portions. These and other hardware provisions have yet to be fully exploited, though clever programming techniques by several implementors have produced real-time shadows and mirrors [Hei91][KSC81].

1.2 Our Approach

Our research expands hardware-based pipeline rendering techniques to present a platform which does provide realism and user interaction, as well as additional means in which to manipulate and comprehend these complex scenes. It proposes evolutionary, not revolutionary, modifications of the graphics pipeline where necessary, and the techniques to use these features for the aforementioned purposes. Although slight hardware modifications may be introduced, many such modifications are based on similar features found in other architectures. This thesis is an introduction of new techniques using these features; it is not an introduction of a new rendering architecture itself. With the high investment in pipeline rendering architectures, better rendering and interactive techniques using these architectures becomes a necessity as the demands of the applications grow.

Our approach to rendering is to take full advantage of the provided graphics rendering pipeline to provide realistic rendering in a dynamic environment. Much of this work is focused on using multi-pass rendering techniques based on existing pipeline architectures. Through these multi-pass methods, we provide a means to include not only reflection but a technique for approximating refractive planar surfaces as well. The model presented extends the current reflection techniques to provide an arbitrary level of refraction and reflection based on bit-plane depth for use in “hall-of-mirror” type environments and to provide a close approximation for refractive objects. An image transform is presented to correct for perspective distortions during the image mapping of the secondary refracted image. For non-refractive transparent surfaces, a display architecture modification is proposed to provide the facilities for correct
sorted surface blending. This extension, the Dual Z-buffer or $DZ$-buffer, along with current frame-buffer functions such as stencil planes and accumulation buffers, provide the hardware platform to render correct transparent surfaces through multiple rendering passes.

Multiple rendering passes also provide the bases for shadow volume support with specular surfaces. We provide a practical shadow volume method which is extended for interaction with specular light reclassification. This multi-pass method is combined with the similar specular surface stenciling methods to provide a recursive methodology which not only preserves shadows in all reflected and refracted images, but which also accounts for refraction and reflection of the light and shadows in the primary and secondary images as well.

Our pipeline rendering platform also includes utilizing hardware provided features such as fog and texture mapping to provide simulation of varying material properties such as translucency and filtering. Fitting of the hardware lighting model and surface attributes to a more physics-based and empirically-derived model further provides more realistic rendering. Combined with the multi-pass features, these techniques provide an alternative to ray tracing for creating fast, approximate specular lighting effects at rates on the order of 5-50 times faster as documented in the examples. We additionally support incorporation of diffuse illumination effects, presenting full scene illumination for dynamic environments. The coordination of these processes is seen in Figure 1.1, with effects demonstrated in Figure 1.2.

Finally, we introduce scene de-cluttering facilities to promote user comprehension of these interactive environments. This includes selective visualization of the environment by supporting arbitrary clipping surfaces in real-time. By combining this with our sorted transparency procedure, the arbitrary clipping surface can be used as an arbitrary transparency surface, making all enclosed areas transparent.
Figure 1.1: Multi-pass Pipeline Rendering Process
Figure 1.2: Multi-pass Pipeline Rendering Image
1.3 Overview

Chapter 2 first introduces the sophisticated illumination effects which traditionally appear only in photo-realistic rendering methods. This includes discussion of the benefits and limitations of each method. Current hardware-based methods used to achieve these effects are then presented.

The next three chapters discuss the primary multi-pass contributions for pipeline rendering of specular environments. Chapter 3 discusses specular surface rendering. It first introduces the standard method of implementing reflection through secondary viewpoints. The case of refractive transparency is then investigated, using an extension of the reflection method for refractions. Non-refractive transparency is additionally supported through techniques which provide correct transparency blending. This includes introducing some required pipeline extensions and support functions, including a Dual Z-buffer. Both back-to-front and front-to-back traversal methods are included. Simulation of scattering material properties such as translucency are also included.

Chapter 4 describes our implementation of shadows which work in conjunction with the reflections and refractions. This includes discussion of a practical implementation of shadow volumes, as well as the various methods of modeling their interaction with specular surfaces. The current implementation’s use of virtual light sources (or light source reclassification) is discussed in a recursive framework, with this method eventually extended to creation of specular light volumes. Material transmission and reflection properties for these light volumes are then described, including simulation of non-uniform surfaces.

Chapter 5 brings the two previous chapters together as a composite recursive procedure. Coordinated recursion of the two multi-pass methods is first detailed, followed by allocation specifics of the primary shared resource, the stencil buffer.

As the previous chapters introduce the primary shadowing techniques, Chapter 6
discusses use of these techniques in producing physically-based global and local illumination effects. This includes global direct effects such as light accumulation and area light sources, local direct effects through fitting of the hardware lighting model, and indirect effects through incorporating a radiosity-generated diffuse-diffuse transfer solution. Both direct and indirect effects are then evaluated in toto.

The performance of all of the previously introduced features are examined in Chapter 7. This focuses on the use of pipeline rendering to bridge the quality/performance gap of traditional rendering. Quality versus timing tradeoffs are discussed in the context of both user-selected criteria and automatic selection in progressive refinement applications.

Whereas the previous chapters focus on the realism of dynamic scenes, Chapter 8 focuses on user comprehension and interaction with these environments. The use of clipping and transparency surfaces is discussed for selective scene rendering. The $DZ$-buffer is used to provide arbitrary clipping surfaces; this is combined with the non-refractive transparency method for providing arbitrary transparency surfaces which, analogous to clipping surfaces, render all enclosed surfaces and volumes transparent.

Chapter 9 addresses the limitations of the system, particularly in regards to the current hardware platform as well as possible extensions to the platform. Other graphics architectures are then examined for relevance, feasibility, and possible extensions to the multi-pass rendering process.

Finally, Chapter 10 summarizes our contributions in presenting the pipeline rendering methodology, and discusses possible future work in this area.
Chapter 2

Background

2.1 Rendering Methods

Traditionally, sophisticated illumination and rendering effects have appeared only in ray-tracing and radiosity systems. This includes reflective specular surfaces, refractive transparent surfaces, shadows and caustics, and translucency. To understand the complexities of these effects, their implementation and limitations in these non-interactive systems will be examined. In addition, existing empirical algorithms for achieving some effects will also be examined.

2.1.1 Basic Recursive Ray Tracing

Although ray-casting was first developed by Appel[App68] and by Goldstein and Nagel[GN71], its use was primarily for hidden surface removal. Appel’s method did determine whether a point was in shadow, but it remained for Whitted[Whi80] to extend ray-casting to ray-tracing to handle reflections and refractions.

In simplest terms, ray-tracing determines the visibility and shading of objects in an environment by tracing imaginary rays of light from the viewer’s eye to the objects. This method casts an eye ray from the center of projection (the viewer’s eye), through a pixel’s center in a window on an arbitrary view plane, and into the
environment. The pixel through which the ray passes has its color set to that of
the first intersected object as determined by the current lighting model. In Appel’s
system, a object’s pixel is in shadow if a surface is between the ray-object point of
intersection and the light source. This is determined by “firing” an additional ray
from this point of intersection to the object and checking for intersections.

Whitted’s extension to Appel’s method fires reflection rays and refraction rays in
addition to Appel’s shadow rays. Reflection rays trace from the point of intersection
in a direction of the incident ray reflected about the surface normal. Refraction rays
trace into the object in a direction determined by the incident ray and Snell’s law.
Each reflection and refraction ray may recursively spawn more reflection, refraction,
and shadow rays. This process is seen in Figure 2.1.

As can be seen with this approach, intersection testing is very important. Much
attention has been paid to reducing the time spent performing intersection checks;
this will be addressed later in the context of our system. As the general nature of ray
tracing is as a non-interactive image generator, we will only focus on the illumination
aspects of ray tracing, not the computation costs involved.

There are many variations of this basic approach which attempt to account for
physical properties of materials and illumination. Many reflection models have been
developed for computer graphics; some are empirical models and some are based
on classic wave theory. The Phong model [Pho75] is the most commonly used re-
fection model in computer graphics, and bases the bidirectional spectral reflectivity
on diffuse and specular coefficients and the viewing angle to the surface. Other
models [Bli77][CT82] generate direct illumination effects using statistical microfacet
approximation for specular effects and a classical Lambertian model for diffuse ef-
ects. Many more complex methods exist, based on light dispersal from and through
an object. These models are too expensive to be investigated in the context of any
hardware-based solution.

Ray tracing from the eye, or forward ray tracing as it is known, has many short-
comings, especially in its model of shadows and refraction. As can be seen in Fig-
ure 2.1, the shadow ray $L_1$ is not refracted on its path to the light because such
refraction would cause it to miss the light source. Because of this deficiency, only
images behind a refractive surface are refracted; light (and any shadow resulting
from that light) passing through the surface is not refracted.

Caustics, the bright overlap of reflected, refracted and primary light rays, are like-
wise impossible in standard forward ray tracing without costly random ray spawning.
This is again due to the inability to fire a ray which is reflected/refracted ray to a
light source.

2.1.2 Backward Ray Tracing

As mentioned above, standard forward ray tracing omits all indirect illumination
except pure speculative components resulting from refraction or reflection to the
light source. Reflection and refraction rays typically miss light sources. This diffuse
interaction is instead approximated by a local Phong reflection and transmission
term. To achieve this and other effects found in radiosity systems, backward ray
tracing was developed. Arvo [Arv86] first suggested this method of casting rays
from the light source in 1986. It has typically been implemented as a two-pass ray-casting technique in several systems [CF87][ZPL88].

The necessity of this two pass approach is seen in the complexity of a solution based on forward ray tracing. To detect these indirect illumination results, enough “feeler” rays would have to be spawned at each point of intersection to have a high probability of detecting illumination from indirect sources. This exponential growth of rays proves extremely prohibitive, and only a few systems have attempted to handle this [Kaj86][War94].

The two pass method obviates these spawned feeler rays by first determining indirect illumination effects by casting rays from the light source. These rays reflect, refract, and intersect with surfaces, producing by spatial density the diffuse illumination of the surfaces. In addition to providing the diffuse illumination of the scene, this process also enables caustics to form where a specular-to-diffuse light transport mechanism takes place, a empirical notion termed by Wallace et al. [WCG87]. Here, light rays both direct and indirect converge and diverge on a surface producing bright and dark patches.

In order to perform this two-step process, illumination effects from the first step must be stored for consideration in the second step. Arvo suggested using an illumination map for each object. Other methods rely not on shooting individual rays, but instead on creation of caustic polygons, or light volumes[WW'92][KG79].

These methods, known as light beam tracing[HH84], cast rays from the light source to each vertex of a polygon of a specular (refractive/reflective) object. Secondary transmitted light rays are created from these vertices in the direction indicated by reflection or refraction to the surface normal. Any intersection of these rays with a diffuse polygon form a caustic polygon to be created on the plane of that polygon. The vertices of the caustic polygon are at the intersection of the transmitted rays from the specular polygon with the plane of the diffuse polygon. Examples of these caustic polygons can be seen in Figure 2.2. During the second rendering phase, the diffuse component of a diffuse polygon is combined with the intensities
of any caustic polygons associated with that polygon. This intensity of the caustic polygon is similar to the form factor from the radiosity method.

2.1.3 Radiosity

Where diffuse illumination is difficult and expensive in ray tracing systems, the nature of radiosity systems is based on calculation of these diffuse interactions through energy transfer between surfaces. Radiosity was first applied to computer graphics by Goral et al. [GTGB84] based on theories of heat transfer between surfaces [SH81]. In radiosity systems, all surfaces are assumed to be Lambertian diffusers, emitters, or reflectors. Surfaces are subdivided into planar “patches” over which the radiosity is constant. The radiosity of each patch is the total rate of energy leaving the surface, which is equal to the sum of the emitted and reflected energies. The reflected energies are the sum of the reflected energies resulting from the individual incident energies on the patch from every other patch, which is derived from a geometric relationship between any two patches known as a “form factor.”

Inherent in the radiosity method are a variety of illumination effects which produce extraordinarily photo-realistic images, including shadows, color bleeding, and
color filtering. The cost of this realism is in very high preprocessing computation and storage requirements for computing the form factors. In addition, the environment is relatively static except for view changes, as any object movement requires recomputation of the form factors. There are systems which have tried to address this static nature, but none have supported a full dynamic environment. The back-buffer extension [BWC86] relies on predefined object paths. Other methods [GSG90],[Che90] rely on propagation of modified form factors in a progressive solution. Even with methods maintaining complex object interactions [FYT94],[MS94], rates are near interactive for only small changes. In addition, the view-independent nature of the radiosity computation usually precludes the support of specular reflection.

2.1.4 Two-Pass Methods

Because radiosity systems handle diffuse components more readily than ray tracing and the converse is true for specular components, these two methods have been combined in another two-pass approach originated by Wallace et al. [WCG87]. In this model diffuse lighting effects are stored implicitly in the final radiosity solution itself during stage one, with view-dependent specularities added through standard distribution ray tracing in the second stage. While producing more physically-realistic images, these two-pass methods suffer from the double cost shortcomings of both methods for a dynamic environment. One noteworthy exception is the image-based rendering techniques [CW93], which sacrifice some image quality for interactive view manipulation as well as some scene dynamics. These systems create intermediate views through interpolation of selected keyframe images.

2.1.5 Beam Tracing

Analogous to ray tracing’s method of casting rays from the eye point and spawning new rays at specular surface intersections is Heckbert and Hanrahan’s method of beam tracing [HH84] which uses pyramidal beams instead of rays. This method
relies on the spatial coherence of neighboring rays; that is, neighboring rays tend to follow the same path.

Beam tracing starts with an initial beam defined by the viewing pyramid. Polygons are sorted from this view using a version of the Weiler-Atherton hidden surface removal algorithm. The view beam's intersection with objects causes reflection and refraction beams to be generated, and the intersecting area to be subtracted from the original beam. As this process proceeds recursively, the view position is updated and the polygons are sorted from the spawned views. An intersection beam tree is created during this recursion, with links representing rays of light (a beam) and nodes representing the surfaces intersected by that beam. The resulting beam tree is recursively rendered using a polygon scan-conversion algorithm.

The beam tracing method has advantages over traditional ray tracing in that it does not spawn rays on a pixel by pixel basis and does not suffer from the aliasing artifacts associated with this individual pixel basis. This method does have several limitations over ray tracing though; foremost is that it does not operate on non-planar surfaces. This is due to the assumption of spatial coherence of the beam. Unlike ray tracing where a single view ray is reflected or refracted, an entire view frustum is bent. This creates a new virtual viewpoint representing the apex of the secondary view pyramid. This is seen in Figure 2.3. The reflected rays of a beam intersecting a curved surface would not intersect at a single virtual viewpoint; therefore, this method is incompatible with curved surfaces. For reflection of planar surfaces, the virtual eye-point can be represented as a linear transform (rotation and translation) of the original view.

The second limitation also stems from this necessity of a virtual eyepoint. Unlike reflection off a planar surface, refraction rays do not converge to a single point; refraction is a nonlinear phenomenon. Rays are refracted according to Snell's Law, which relates the incident and refracted angles: \( \eta_1 \sin(\theta_1) = \eta_2 \sin(\theta_2) \). In beam tracing, refraction is approximated with a linear transformation. This transformation is described in Appendix B.
Figure 2.3: Virtual viewpoint for reflected beam

Figure 2.4: Z-buffer based rendering pipeline

2.1.6 Hardware-based Rendering

While sophisticated illumination effects have been achieved in radiosity and ray tracing systems, these effects are achieved with significant precomputation overhead in relatively static environments. Most graphics hardware systems provide only empirically-based Phong shading, an ambient term to approximate diffuse interactions, and alpha blending to simulate partial transparency. Very few of the complex illumination effects have been achieved using the graphics hardware architectures provided by most workstation manufacturers, in essence forfeiting the advanced pipeline features for software calculations and off-line processing. A typical hardware rendering pipeline is seen in Figure 2.4.
One exception to this is the increased use of shadows in real-time systems. Real-time shadows have been achieved using hardware-provided stencil planes [Hei91] as well as through the use of projective textures [SKvW+92]. Of course, these implementations are also susceptible to all of the usual image-space aliasing problems such as missing pixels at coincident polygon edges or texture magnification filtering.

Heidmann uses a hardware implementation of Brotman and Badler’s[BB84] variation of the shadow volume technique proposed by Crow[Cro77]. This method generates shadow polygons for each silhouette edge of the object. These shadow polygons are invisible polygons which extend from the edge away from the light. For every pixel in the scene, the stencil buffer is used to keep track of how many shadow polygons are between the viewer and the pixel’s object. If the number is odd, the pixel is in the middle of a shadow volume and therefore in shadow. If the value is even, the pixel is not inside a shadow volume and therefore lit. This process can be seen in Figure 2.5.
Projective textures were originally introduced by Williams [Wil78] and have been recently implemented using texture mapping hardware [SKvW92]. This method creates a light-view image of the scene and uses this as a projected texture in the environment.

In addition to the hardware provisions for shadows, non-refractive transparency has been loosely supported through the use of alpha-blending. This method is incompatible with Z-buffer sorting, however, and no provision is made for refractive transparency. Non-refractive transparency methods are addressed in Chapter 3.

Other approximations for illumination effects have been introduced such as reflection effects using cubic environment maps to create specular highlights [VF94]. Such methods generally rely on creation of these effects without regard to the other objects in the environment; i.e. reflections are based on an image, not the surrounding objects.

The following chapters describe new rendering and illumination techniques which are based on and use standard graphics hardware architectures. These effects are recalculated for each frame at interactive to near-interactive rates. All described functionality has been implemented using existing SGI hardware-based graphics support.

## 2.2 Definitions

For the purposes of this discussion, we shall introduce terms common to users in the GL environment and some describing implementation techniques.

### 2.2.1 Stencil Planes

*Stencil planes* are essentially an enhanced Z-buffer introduced (but not named) by Brotman and Badler [BB84]. In its simplest form, pixels are written only if the current stencil value (analogous to the current Z value) of the destination pixel passes the
defined stencil test. Depending on the result, the pixel is written and the stencil value is changed. Stencil plane calls take the form:

\[ \text{stencil} \left( \text{COMP\_FUNC}, \text{COMP\_VALUE}, \text{PASS\_FUNC}, \text{PASS\_VALUE} \right) \]

where

- **COMP\_FUNC** The compare function for the stencil operation to pass. The current pixel’s stencil value is compared against the COMP\_VALUE using this function to return the boolean result. Choices are (EQUAL, GREATER, LESS, GTEQUAL, LTEQUAL, NONE).

- **COMP\_VALUE** The value which the current pixel’s stencil value is compared with using the COMP\_FUNC.

- **PASS\_FUNC** The function which is applied to the current pixel’s stencil value using the PASS\_VALUE as the parameter. Choices are (REPLACE, CLEAR, INC, DEC, KEEP).

- **PASS\_VALUE** The value used to update the current pixel’s stencil value according to the chosen PASS\_FUNC.

### 2.2.2 Accumulation Buffer

An *accumulation buffer* [Car84] is a secondary image buffer to which the current image can be added. The resulting image can also be divided by a constant. This enables a blending of images or image features. Accumulation buffer calls are in the form:

\[ \text{accbuf} \left( \text{OPERATION} \right) \]

where OPERATION is one of the following:
### ACC_CLEAR
Clear the accumulation buffer

### ACC_ACCUMULATE
Add the contents of the current framebuffer to the contents of the accumulation buffer.

### ACC_RETURN
Copy the contents of the accumulation buffer back to the current framebuffer.

#### 2.2.3 Alpha Blending

Blending of the to-be-drawn pixel’s RGBA components with existing values is accomplished using the blend function. The format of this is:

\[
\text{blendfunction}(\text{sfactor}, \text{dfactor})
\]

where sfactor and dfactor are the blending factors by which to scale the source and destination pixel values, \(C_s\) and \(C_d\), respectively. Each RGBA component is determined by the following specification:

\[
C_d = C_s \cdot \text{SFACCTOR} + C_d \cdot \text{DFACCTOR}
\]

where SFACCTOR and DFACCTOR are one of the following choices:

- BF_SA (source alpha)
- BF_MSA (1-source alpha)
- BF_DA (destination alpha)
- BF_MSA (1-destination alpha)
- BF_ONE (1)
- BF_ZERO (0)

#### 2.2.4 Shadow Volumes

*Shadow volumes* are volumes of shadow cast by opaque objects. For polygonal objects, the shadow volume is comprised of silhouette faces generated from the object’s
silhouette edges. A silhouette edge is an edge which divides a lit (facing light) face and an unlit (facing away from light) face. A silhouette face is a face created for each silhouette edge of an object by extending that edge away from the light source along the light-ray direction. Pixels inside the volume are in shadow; pixels outside are lit.

2.2.5 Light Volumes

Light volumes are volumes of light bounded by silhouette faces of reflecting and refracting objects. As with shadow volumes, silhouette faces are created for each edge of a specular object by extending that edge away from the virtual light source position along the light-ray direction.

2.2.6 In-Out Refractions

In-out refractions are refractions which occur when light passes from one medium to another and back to the first, such as light traversing through a piece of glass. There is an entry refraction and an exit refraction, producing a refracted ray parallel to the incident ray in surfaces where the in-out surfaces are parallel (such as a sheet of glass).
Chapter 3

Specular Surface Rendering

Specular surfaces (surfaces which exhibit specular reflection or transmission) are commonplace in many real-world environments. This ranges from the presence of reflective mirrors and transparent refractive water to partially specular surfaces such as a shiny wet floor or frosted glass. While reflective and refractive surfaces have been supported in ray tracing since their introduction by Whitted [Whi80], only non-refractive partially-transparent surfaces have been readily available in hardware-based graphics. Each of these specular surface types is further examined in regards to a multi-pass hardware-based pipeline methodology.

3.1 Reflections

Reflective images have been generated in ray tracing systems by tracing individual reflection rays which are spawned from intersections with reflective surfaces. Additionally, a reflective image can be seen as corresponding to an inverted image from a secondary viewpoint. In other words, the reflected image is the flipped image from a viewpoint on the “other” side of the mirror. This analogy provides the basis for planar mirror reflection in several hardware-based systems [KSC81][HH84].

In multi-pass pipeline rendering, mirrors are implemented by rendering the entire environment, exclusive of the mirrored surface. The mirrored surface is drawn with
Z-buffering, creating a stencil mask of pixels where the mirror is visible. A second rendering of the environment is then performed from the reflected viewpoint, drawing only over the previously masked pixels. Because the reflected angle (angle from mirror plane to reflected viewpoint) is the negative of the incident angle and because the image is flipped, the reflected image directly “fits” onto the mirror.

The calculation of the virtual camera position follows from reflection of the incident line of sight with the plane on which the specular surface lies. Not only does this result in a virtual camera position, but this transform involves scaling the scene about the $Y$ axis to flip or mirror the image. This scaling is implicit in the transform given in [HH84] which is derived from the equation representing reflected points $P_r$ in terms of the original points $P_i$ and the plane equation $LP$ having normal $N$. As $LP$ also gives the distance from the plane for any point, $P_r = P_i - 2(LP_i)N$ expresses in vector form the transformation involved. This transform as expressed as a homogeneous 4x4 matrix is included in Appendix B.

The virtual viewpoint transform is applied during a second rendering of the environment. This second rendering is performed only in the area masked during the first pass. Before this second rendering is performed, the $Z$ values must be cleared in this masked area. As the necessary area is already masked, rendering a polygon with the appropriate $Z$ coordinates and with $ZF\text{ALWAYS}$ as the compare function resets the $Z$ values for the specular surface. The second rendering can then be performed using the normal Z-buffer depth sorting. All necessary viewpoint culling can also be applied from this virtual viewpoint.

In addition, this process can be repeated recursively when multiple reflective surfaces exist. The virtual viewpoint gets transformed for each specular-specular transport, thereby producing reflections of reflections to a chosen depth. This process is demonstrated in Figure 3.1 for three specular surfaces at various recursion depths. The recursive process and accompanying stenciling method are detailed in Chapter 5.
Figure 3.1: Recursive Reflection
As in beam tracing, the above described virtual viewpoint method also relies on planar reflective surfaces. Again, this is due to a single reflected virtual viewpoint resulting from a linear transformation of the original eye point. Where a non-linear transformation is involved, no single virtual viewpoint exists and an approximate viewpoint must be selected. Such an approximation is required for refractive planar surfaces.

### 3.2 Refractive Transparency

Refractive transparency has generally only been available in ray tracing. Ray tracing implementations are based on one of several illumination models, both empirical and physically based. Hall [HG83] introduced an extension of the Phong reflection model for the transmission term. As with the Phong reflectance model, this model accounts for the spread of transmitted light through a refractive medium. Distribution ray tracing [CPC84] supports blurred refractions through through a jittered sampling of refraction rays. Attenuation of transmitted light usually occurs based on Fresnel transmission coefficients. These applications support translucent as well as transparent materials.

While refractive transparency has been unknown in hardware-based rendering, we can provide the refractive surface rendering itself using a method analogous to the reflection method described in Section 3.1. Although refractive images are similar in concept to reflections, they are more complex in practice.

Whereas a mirrored image directly corresponds to the reflective surface to which it maps, a refracted image maps to a distorted image space. Simply performing a second rendering in the stenciled area does not overlay the correct image portion. This is demonstrated in Figure 3.2. The area visible through the transparent surface in the refracted view is different than the image area from the original viewpoint; areas outside the refracting surface and even in front may be visible in the refracted image. This difference is due to two factors; the difference between incident and
refracted viewpoints and the perspective distortion.

Because the incident angle does not equal the refracted angle, the refracted image is rotated with respect to the original image. This is further compounded by the rotated image plane undergoing a perspective distortion different than the perspective distortion of the original plane. The perspective transformations are the same, but because the planes have different orientations, the resulting distortions are different. The result is that a refractive square planar face, for example, maps to two different quadrilaterals in the original versus the refracted images.

The refractive image \( I_R \) does correspond to the original image \( I_O \) through a 3x3 2-D bijective projective mapping \( M_3 \). This mapping is the intersection of the 4x4 3-D image mapping set \( M_4 \) with the reflective planar surface \( \Psi \):
\[ \mathbf{I}_0 = \mathbf{I}_r \mathbf{M}_3 \]  \hspace{1cm} (3.2) \\

where \\

\[ \mathbf{M}_3 = \frac{1}{3} \mathbf{M}_4, \]  \hspace{1cm} (3.3) \\

and \\

\[ \mathbf{M}_4 = \mathbf{P}^{-1} \mathbf{C}_r \mathbf{C}_o^{-1} \mathbf{P}. \]  \hspace{1cm} (3.4) \\

In equation 3.4, \( \mathbf{P} \) is the perspective transform and \( \mathbf{C}_o \) and \( \mathbf{C}_r \) are the original and refracted camera transforms, respectively.

This results in a 2-D projective transform of arbitrary quadrilateral to quadrilateral described in [Hec89] and included in Appendix A. This transform, described by a 3x3 homogeneous matrix, can be applied directly to the screen-viewport mapping to distort the refractive image into the normal image space. In hardware which supports user-defined operations, this transform can be inserted directly at the end of the rendering pipeline. In systems where this is not possible, such as the Silicon Graphics architecture, this transform can be implemented as a 4x4 homogeneous transform inserted in the world-to-unit pipeline. The resulting transform is constructed with a zero scale factor for \( Z \) so that the mapping is to the \( Z = 0 \) plane.
Without this mapping, the tapering and skewing effects from the quadrilateral distortion affect the Z coordinates. Unlike the 2-D transform, the 3-D does, however, preclude the use of the Z-buffer for hidden surface removal as all image points now have the same Z value. This method also does not allow for the fog translucency simulation described in Section 3.4, due to the loss of depth.

Note also that this method does not produce true refractions, merely a close approximation to the refractive image. In a true refractive image, every ray incident with the refractive plane bends according to its angle with the plane and Snell’s Law; this method, as in beam tracing, uses only one incident angle. In practice, two angles are used to provide more realistic results with the system. First, the incident ray is taken from the camera location to the refracting face center to determine whether the incident angle is greater than the critical angle. If this is the case, the surface is taken to be wholly reflective. If the angle is less than the critical angle, the incident angle for Snell’s Law is taken at the point of intersection of the view vector (camera’s negative Z axis) and the plane in which the refracting face lies. This method insures that the critical angle is reached as the plane moves tangentially to the view, yet the refracted image is seen as a smooth scrolling of the background behind the face.

In the original implementation of this work, the virtual camera position was determined by refraction or rotation of the original camera position around the above described point of intersection according to Snell’s Law. While this approximated the bending of light along that ray, it is not the best approximation for the distortion which takes place due to the varied refraction of individual rays. In practice, a refracted image appears at \( n \) times the actual distance from the refracting medium whose index of refraction is \( n \). As noted by Heckbert and Hanrahan[HH84] and seen in Figure 3.4, this approximates for paraxial rays to a parallel displacement of the viewpoint to the plane, termed the Tangent Law. Expressed in vector form, we see the equation

\[
P_r = P_i + (n - 1)(L P_i) N
\]  
(3.5)
and its corresponding 4x4 transform given in Appendix B is analogous to the reflection equation presented above. In fact, the reflection is simply the special instance where \( n = -1 \).

Figure 3.4: Tangent Law for paraxial rays

Figure 3.5: Uncorrected Refraction

The contrast between the two methods of determining the refractive viewpoint
can be seen in Figure 3.5, where image (b) is based on paraxial displacement. As can be seen, this method further necessitates the use of the detailed projective transform as clipping discrepancies become far more apparent. The corrected images are seen in Figure 3.6.

Figure 3.6: Corrected Refraction

Because of Tangent Law’s closer approximation for paraxial rays, our original implementation was therefore modified to include this Tangent Law transform. Again, two angles are used to account for tangential movement of the refracting plane. While this method is more accurate than the Snell Law implementation, it is also simply a close approximation. Other such approximations might be possible based on further viewport manipulation such as manipulation of the field of view rather than displacement of the viewpoint. A more flexible rendering pipeline providing additional vertex and image transformation control would enable greater accuracy in simulating such phenomena.

Note also that this recursive method provides automatic sorting of transparent faces for the alpha blending. Blended transparency requires surfaces to be drawn in sorted order, a feature not supported by current Z-buffer architectures. The recursive nature of the traversal dictates that transparency blending occurs after the refracted image containing other transparent objects has been rendered.
3.3 Non-Refractive Transparency

Transparency in hardware based systems is almost always non-refractive transparency (with an exception first noted in [DB94]), and many times this approximation over refractive transparency suffices. Most Z-buffer-based systems support screen-door (dithered) transparency or simply render transparent polygons last using alpha blending based on interpolated transparency [FvDFH90]. If the transparent polygons are not depth-sorted, the resulting image is incorrect. There are other issues and approaches; the following summary details existing methods.

Screen-Door Transparency

Screen-door or dithered transparency uses a mask to implement a mesh of the transparent object’s pixels. Pixels are only written if their corresponding position in a transparency bit mask is set. Spatial proximity of neighboring pixels results in color blending producing interpolated values.

There are many problems with screen-door transparency, foremost being mask conflicts and dithering artifacts. Two transparent objects cannot share a mask or one will completely overwrite the other. In addition, extensive masking results in noticeable dithering effects, producing an undesirable artifact pattern on the objects. Even subpixel algorithms are not accurate for multiple transparency layers [Ake93b].

Hardware Sorted Transparency

Most Z-buffer based systems rely on an empirical approach to transmitted light attenuation. Non-refractive transparency is approximated using a linear transparency parameter $t$ to blend the object pixel intensity $I_o$ and the background pixel intensity $I_b$ in $Z$ space using the combination

$$I = tI_o + (1 - t)I_b, \quad 0 \leq t \leq 1$$  \hspace{1cm} (3.6)

Although this formulation requires an ordered depth blending (back to front) of
overlapping transparent pixels, Z-buffered sorting does not provide this facility. In addition, presorting the transparent surfaces suffers from all the traditional hidden surface removal problems such as intersecting surfaces and the need to find a (possibly non-existent) correct sort order (e.g. [NNS72]).

In [Mam89], Mammen describes a method which renders the transparent objects in the correct back-to-front order without presorting. To accomplish this, blending occurs at the pixel level in a series of iterations. At each iteration over the transparent set, the transparent pixels closest to the opaque pixels are determined and blended in with the opaque value. This farthest transparent pixel replaces the opaque pixel for the next iteration.

This method relies on secondary sort buffers used to store depth and tag information; an opaque pixel map initially maintains the opaque rendered objects. At each pass, every transparent pixel which is processed has its depth compared with the stored opaque depth and the stored sort depth. If the pixel’s depth is greater than the sort depth and in front of the opaque depth, that pixel’s information is stored in the transparency buffers. This information includes the depth, color, and alpha values. After all pixels have been processed, opaque colors and depths are updated on a pixel by pixel basis using the resulting transparency buffers. This method is repeated for the number of transparency layers.

Mammen also introduced a variation of this method in which only the depth transparency buffer is required. Two passes are made at each iteration; the first writes depth information which acts as a tag for the second pass. During the second pass, the scene is rendered and color is blended with the opaque buffer. Each stage requires Z-buffer sorting.

Mammen fails to note the successive reduction of transparency area which permits a simple swap of Z-buffers, and his method suffers from certain depth problems described in Section 3.3.3. He mentions other applications of this method, but he does not observe that he can use his two-buffer technique for clipping.

Kelley[KGP+94] introduces a hybrid of Mammen’s method and a simple list sort.

35
This method uses a multiple layered Z buffer (2, 4 or 8) which maintains the four closest visible layers per pixel. If overflow of the buffers occur, layers are composited into one layer and the process continues with three free layers. This method has much additional storage requirements, as well as costs associated with compositing of the buffers during overflow.

Other multiple buffer techniques exist [S89a][WHG84] which rely on Z-associated information, but these are too expensive to act as a low-cost interactive solution.

### 3.3.1 Definitions

**Dual Z-Buffer**

A Dual Z-buffer, or DZ-buffer, consists of two distinct but functionally equivalent Z-buffer areas each with its own compare function. One is designated as the current write buffer. Depth comparisons must pass both buffers’ designated test in order for the pixel’s Z-value to be written to the designated write area. With the two buffers designated Z1 and Z2, a sample configuration might be:

```c
zfunction(z1, ZF_LESS);
zfunction(z2, ZF_GTEQUAL);
zwrite(z1);
```

providing depth sorting of each subsequent write to buffer z1 which has greater or equal depth than the contents of buffer z2.

An extension of the DZ-buffer concept is the Tri Z-buffer, or TZ-buffer which presents a third functionally equivalent Z-buffer for depth comparison.

**Accumulation Buffer**

A modified accumulation buffer is presented in this section in which the return operation does not perform a pure copy. Instead, this operation acts as a typical
framebuffer operation and makes use of the blendfunction. This can be readily simulated using the current implementation and directread and directwrite buffer functions which read and write the framebuffer area using the designated blending operations.

**ACC_RETURN** Blend the contents of the accumulation buffer back to the current framebuffer according to the current blendfunction.

**Alpha Blending**

A modified blending function is also presented which provides the facilities necessary for implementing the front-to-back transparency blending. The additional SFACTOR choice is:

\[ \text{BF\_SAxBF\_MDA} \quad \text{source alpha} \times (1 - \text{destination alpha}) \]

which is necessary to implement the transparency blending equation 3.9.

The blendfunction is also split into two blending components, the RGB color components and the A transmittance component. This is necessary since the transparency blend equation is not symmetrical for the color and alpha components. The two new functions are:

\[ \text{cblendfunction(sfactor, dfactor)} \]
\[ \text{ablenfunction(sfactor, dfactor)} \]

whose use is identical to the existing blendfunction for color and alpha components, respectively.

**3.3.2 Back-to-Front Transparency**

Partially transparent surfaces can be rendered correctly using the DZ buffer in a back-to-front manner similar to [Mam89]. After all fully-opaque objects have been rendered, all pixels of the transparent objects which are visible yet furthest away
are rendered. A second iteration over the transparent objects then renders all pixels second furthest away. This process repeats to the depth of visibly overlapping transparent surfaces, each iteration rendering the next closest pixel with the appropriate alpha blending. This process insures that the transparent objects are blended in the correct order, demonstrated in Figure 3.7.

![Figure 3.7: Back-to-front rendered area](image.png)

The use of the $DZ$-buffer in this method is seen in each iteration. At each iteration, one $Z$-buffer is designated as PREVIOUS, one is designated as CURRENT. The buffer to be written to is CURRENT. $Z$ comparisons must be closer than PREVIOUS, yet further than CURRENT in order to be written to CURRENT. As CURRENT gets updated during this iteration, only the furthest pixels which are closer than the last iteration are written.
draw_scene( )
{
    PREVIOUS=1; CURRENT=2;
zclear(PREVIOUS,ZMIN); //init z values
zclear(CURRENT,ZMAX);
zwite(CURRENT);
zfunction(CURRENT,ZF_LESS);
zfunction(PREVIOUS,ZF_NONE); //normal Z-buffer operation
draw_opaque_objects( ); //draw opaque with color
blendfunction(BF_SA,BF_MSA); //alpha blending
do{
    temp=PREVIOUS; //swap buffers
    PREVIOUS=CURRENT;
    CURRENT=PREVIOUS;
btf(PREVIOUS,CURRENT);
}while (something was drawn)
}
btf(PREVIOUS,CURRENT)
{
zfunction(CURRENT,ZF_GREATER); //furthest and...
zfunction(PREVIOUS,ZF_LESS); //...closer than last
zwrite(CURRENT);
if (TWOPASS) //if using only one color buffer
    wmpack(0x00000000); // then disable color
for (i=0; i<nsurfs; i++){
    loop through surfaces
    stencil(NONE, ZERO, REPLACE, i);
    draw_surface(i); //Z sort, further than CLIPBUF
}
if (TWOPASS){
    wmpack(0xffffff ff); //enable color
    zfunction(CURRENT,ZF_NONE); //turn off Z-buffering
    zfunction(PREVIOUS,ZF_NONE);
    for (i=0; i<nsurfs; i++){
        stencil(EQUAL, i, KEEP, STEN_VALUE);
        draw_surface(i); //Z sort, further than CLIPBUF
    }
}
}

At each iteration, two passes must be made of the transparent objects. The
first pass does not write color information; its purpose is to compare and write
depth values and tag the stencil buffer with an identifier of the surface to draw in
the second pass. Where Mammen uses depth values to determine which objects to
render in his two pass variation, the process here uses the stencil mask during the
second pass to render the colors where indicated. An object is only rendered in the
second pass at pixels which were tagged in the first pass.
An extension to this method requiring only one pass is possible using the modified accumulation functionality as described above. Color information can be written to a second image buffer with pixel blending set for overwrite. This generates an image of that particular level of transparency, which is then blended with the final image based on transmittance values.

In each method, alpha values are used as a transmittance value for the alpha blending function

\[
I_{\lambda D} = (1 - k_{tS})I_{\lambda S} + k_{tS}I_{\lambda D}
\]

where \(I_{\lambda D}\) and \(I_{\lambda S}\) are the source and destination intensities and \(k_{tS}\) is the source object’s transparency.

The process between depth rendering iterations is what further differentiates our approach from a pixel iteration method such as Mammen’s and provides the low cost and high performance hardware and feasibility. Whereas Mammen relies on individual comparison and copying of \(Z\)-values from the transparency \(Z\)-buffer to the opaque \(Z\)-buffer, this method uses a simple buffer switch. Mammen and similar methods rely on copying the updated \(Z\) values from each iteration to a single buffer for maintaining a complete list of all rendered \(Z\) values. With the \(DZ\)-buffer, the CURRENT and PREVIOUS buffers are simply switched and the CURRENT buffer is cleared, so that on the next iteration all pixel values must be closer than those just written. This is accomplished by switching the two \(Z\)-functions assigned to each buffer as well as by switching the designated write buffer. Costs are minimized as one \(Z\)-buffer must be cleared only one time for each iteration, and no pixel-by-pixel operations are necessary.

With the buffer switch method, only the pixels’ \(Z\) values which were rendered during that iteration are present for comparison in the next iteration since the CURRENT buffer is cleared at the beginning of each iteration. Maintaining only the previous iteration’s \(Z\) values is possible by noting that in all later iterations, the transparent surfaces which need to be rendered will fall in the screen area of those
already rendered. By limiting the draw area of the screen to only the area which was drawn during the last iteration, re-rendering of a transparent surface during a later iteration is eliminated. This limiting process is similar in concept to Weiler and Atherton’s hidden surface removal technique [WA77]; limiting the rendering area is accomplished using the stencil mask which permits a practical implementation.

One method of accomplishing this masking is the following stencil function. At each iteration, the valid rendering area contains stencil values equal to that iteration number. During that iteration’s rendering of the transparent surfaces, those pixels’ stencil values are incremented by one. These become the valid rendering area during the next iteration. This prevents an area drawn in an earlier iteration from being drawn again, as the viable drawing area effectively shrinks with each iteration. With this shrinking, the simple Z-buffer switch becomes feasible. While this stencil function is one of the most simple, there are other stencil methods which are not limited by the stencil buffer depth.
3.3.3 Front-to-Back Transparency

While correct sorted transparency is often a requirement for visualization, approximation often suffices at times in interactive environments. For example, many walkthrough systems do not render in full detail while the view is changing; resolution increases as soon as the user stops traversal. In a situation such as this, complete sorting and display of all transparent surfaces may not be necessary at all times; only a correct looking approximation is desired.

While the above method does provide correct rendering of transparent surfaces, it is highly dependent on the number of overlapping transparent surfaces to achieve a correct appearing image. Since the closest of overlapping surfaces is rendered last, it is not until the last iteration over the transparent surfaces that a correct appearing image is produced. Until the correct number of iterations is reached, surfaces may appear to have holes in them which show transparent segments beneath. This section describes a technique which renders the transparent surfaces front-to-back, thereby displaying the closest transparent segments first, and successively adding additional transparency which appears behind the currently displayed transparent objects. This produces a more correct looking image at each iteration, with no transparent surfaces ever demonstrating the “hole” effect. This method can provide a realistic looking image even if the correct depth is not reached, and the contrast between the two approaches can be readily seen at each iteration in Figs. 3.8 and 3.9.

The front-to-back rendering method uses the $DZ$-buffer analogous to the reverse of the back-to-front method, except the blending function and iteration initialization are different. In the back-to-front method all opaque objects are first rendered, and the transparent pixels are layered on top. In the front-to-back method, opaque objects are rendered with no color to initialize the $Z$-buffer at each iteration (which can be reduced to one iteration as detailed below), and a final color rendering is blended only at the end of the process. This second rendering is unnecessary if two framebuffers such as in double-buffering are available, as the color can be rendered
initially and subsequent transparency renderings can be made to the other frame-buffer. The final blending of opaque and transparent buffers can be accomplished using a modified accumulation buffer.

In the first iteration, all pixels of the transparent objects which are visible and closest to the viewer are rendered. A second iteration over the transparent objects then renders all pixels second closest. This process repeats to the depth of visibly overlapping transparent surfaces, each iteration rendering the next furthest pixel with the appropriate alpha blending. As with the back-to-front method, this can be a two pass process using the stencil planes or a one pass process using a second image buffer.

Again, the use of the DZ-buffer in this method is seen in each iteration. At each iteration, one Z-buffer is designated as PREVIOUS, one is designated as CURRENT. The buffer to be written to is CURRENT. Z comparisons must be closer than CURRENT, yet further than PREVIOUS in order to be written to CURRENT. As CURRENT gets updated during this iteration, only the closest pixels which are further than the last iteration are written.
draw_scene()
{
  accbuf(ACC_CLEAR); //clear accumulation buffer
  PREVIOUS=1;
  CURRENT=2;
  zclear(PREVIOUS,ZMIN); //init z values
  zclear(CURRENT,ZMAX);
  zwrite(CURRENT);
  zfucntion(CURRENT,ZF_LESS);
  zfucntion(PREVIOUS,ZF_NONE); //normal Z-buffer operation
  blendfunction(BF_ONE,BF_ZERO); //normal opaque drawing
  draw_opaque_objects(); //draw opaque with color
  accbuf(ACC_ACCUMULATE); //save image
  clear(0); //clear screen
  cblendfunction(BF_DAxBF_MSA,BF_ONE); //color-alpha blending
  ablendfunction(BF_ZERO,BF_MSA); //alpha-alpha blending
  do{
    ftb(PREVIOUS,CURRENT); //swap buffers
    temp=PREVIOUS;
    PREVIOUS=CURRENT;
    CURRENT=PREVIOUS;
  }while (something drawn)
  blendfunction(BF_DA,BF_ONE); //set final blend
  accbuf(ACC_RETURN); //blend in opaque image
}

ftb(PREVIOUS,CURRENT)
{
  zfucntion(CURRENT,ZF_LESS); //closest and...
  zfucntion(PREVIOUS,ZF_GREATER); //...further than last
  zwrite(CURRENT);
  wmpack(0x00000000); //disable color
  draw_opaque_objects(); //initialize z values
  if (TWOPASS) //if using only one color buffer
    wmpack(0x00000000); //then disable color
  else //else
    wmpack(0xffffffff); //enable color
    for (i=0; i<nsurfs; i++){
      stencil(NONE, ZERO, REPLACE, i); //loop through surfaces
      draw_surface(i); //Z sorted, further than PREVIOUS
    }
  wmpack(0xffffffff); //enable color
  if (TWOPASS){
    zfucntion(CURRENT,ZF_NONE); //turn off Z-buffering
    zfucntion(PREVIOUS,ZF_NONE);
    for (i=0; i<nsurfs; i++){
      stencil(EQUAL, i, KEEP, STEN_VALUE);
      draw_surface(i);
    }
  }
}
Like the back-to-front method, two passes must be made of the transparent objects at each iteration. Alpha values are used as a transmission coefficient for the blending function (which is not supported in the current Silicon Graphics GL definition):

\[
I_{AD} = I_{AD} + k_{tD}(1 - k_{tS})I_{AS}
\]

\[
k_{tD} = k_{tD}k_{tS}
\]

where the destination transparency \(k_{tD}\) is updated at every iteration.

After the transparent surfaces have been rendered, they are blended with the opaque rendered surfaces using a variation of the interpolated transparency equation. As the colors have already been multiplied by the transmission values during the iterative steps, the resulting final blend becomes

\[
I_{AD} = I_{AS} + k_{tS}I_{AD}
\]

This is accomplished through use of the modified accumulation buffer which permits blending on the return mode.

As can be seen in the above description, the opaque objects are re-rendered at each transparency iteration to re-initialize the CURRENT Z-buffer. This can be a very costly process in typical environments which have numerous opaque polygons. This costly step can be eliminated by noting that the Z values produced are the same at each iteration, and therefore need only be rendered once and stored. While this can be accomplished with a buffer copy at each iteration, the buffer transfer is typically expensive without specialized hardware. An extension of the \(DZ\)-buffer to include a third buffer, creating a \(Tri\ Z\)-Buffer or \(TZ\)-buffer, obviates this buffer transfer. This third buffer, named STATIC for our purposes, is used as the write buffer for the initial opaque rendering. The method then proceeds as above, with the addition of a third Z compare function

\[
\text{zfunction}(\text{STATIC}, \text{ZF}_\text{LESS});
\]
replacing the rendering of the opaque objects within the front-to-back sorting function. This third zfunction additionally compares the transparency $Z$ values against the opaque values at each iteration.

### 3.4 Translucency

Refraction and reflection need not be limited to purely transparent or purely specular surfaces, respectively. We can create a multitude of materials by rendering the refractive surface again after the refractive image is drawn. This second rendering is alpha-blended with the refracted scene. On systems such as the Iris Indigo$^\text{TM}$ system, this actually requires two additional renderings of the refractive face since lit faces cannot have a source alpha value. The first rendering is done without color and sets the destination alpha value to the appropriate value. The second rendering is with lighting and a blending function depending on the destination pixel chosen. This permits hardware shading effects and other hardware rendering features such as textures to be blended with the refracted scene. By adjusting the alpha blending values, the “shininess” or specularity of the material can be controlled. Coordinated with texturing, materials such as polished marble or wet tile can be simulated, as seen in Figure 3.10.

![Figure 3.10: Partial Specular Image](image)

(a) Dull Floor  
(b) Shiny Floor

In addition, translucent and other light dispersing materials can be simulated
using the hardware fog feature and the stencil planes. Translucent objects act as a filter with closer objects more clearly visible than farther objects due to the random refractions which take place [KG79]. This effect can be approximated using hardware fog features with the minimum fog set at the refractive plane distance and the maximum at the desired distance depending on the material property. Although fog is linear with respect to the view, the approximation is many times fairly accurate due to the limited angular displacement of the refracting plane because of the critical angle. A more versatile fog function supporting a general linear transform would provide much more accurate translucency and light scattering for reflective surfaces. By incorporating multisampled stochastic \((X, Y)\) shearing about the specular surfaces normal axis, light scattering through the translucent material is additionally simulated. This is accomplished by accumulating using an accumulation buffer [HA90]-intermediate stochastically-sheared specular images to produce the final scattered effect.

A 4x4 transform is created which includes a stochastically generated \((X, Y)\) shear. This transform is premultiplied by the global inverse transform of the specular face normal and postmultiplied by the global transform of the normal, with the resulting transform pushed onto the view matrix stack. This creates a shear linear with respect to the perpendicular distance from the specular surface.

An overview of the entire rendering process is seen in the following code.

```cpp
mask_face(spec_face);    // set stencil area for face
reclassify_camera(Camera); // move camera to virtual viewpoint
enable_clip_plane(spec_face);  // clip geometry on wrong side of face
for (i=0; i<numsamples;)
    shear_view(Camera, spec_face, sample[i]); // shear view stochastically around specular face normal
draw_window(Camera);       // recursively draw reflected/refracted view
accbuf(ACCUMULATE);        // accumulate intermediate image
}
disable_clip_plane(spec_face);    // turn off clipping plane
accbuf(RETURN);                // display composite image
```
By adjusting the jitter amount and the fog parameters, this method can provide a range of effects similar to those produced by analytic methods\cite{Arv95}, although at a much lower cost. Figure 3.4 compares analytically-generated images\footnote{Butterfly design by Elsa Schmid\cite{Sch69a}.} (a) and (b) with our multi-pass images (c) and (d). Our frosted glass images were each generated in less than 0.1 seconds, as compared to 6.5 seconds in the analytic approach. Figure 3.12 compares the two approaches for a scattering reflective surface. Here, analytically-computed image (a) required 1-2 minutes of rendering time, as compared to under 0.5 seconds for multi-pass images (b) and (c).

Figure 3 demonstrates the dispersing nature of a translucent scattering surface under varying shear multipliers (m) and fog parameters (fmax). Note how the elongated blue beam is clearer the closer it is to the glass. Figure 4 again demonstrates this effect in conjunction with texture mapping to simulate a shiny marble surface. As this method uses hardware-based rendering, image size has minimal effect on timings in contrast to ray tracing. In systems where rasterization is independent of polygon size (i.e. Pixel-Planes\cite{FP81}), image size is not a consideration.
(a) Low-Scattering Glass (Analytic)  (b) High-Scattering Glass (Analytic)

(c) Low-Scattering Glass (Multi-Pass)  (d) High-Scattering Glass (Multi-Pass)

Figure 3.11: Frosted Glass
Figure 3.12: Glossy Table
Figure 3. Frosted Glass

(a) $m=0.3$, $f_{\text{max}}=500$

(b) $m=0.5$, $f_{\text{max}}=500$

(c) $m=0.5$, $f_{\text{max}}=300$

(d) $m=0.9$, $f_{\text{max}}=150$

Figure 4. Glossy Marble Wall
Chapter 4

Shadow and Light Volumes

Whereas the previous chapter discussed one of the characteristic features usually found only in ray tracing, namely specular image generation, this chapter addresses another. This feature is shadowing resulting both from direct and specular illumination.

4.1 Shadow Volumes

Our shadows are implemented based on Brotman and Badler’s[BB84] extension of Crow’s shadow volume method[Cro77]. This technique uses the plus-minus principle of silhouette faces to mask regions inside the shadow volume. The use of shadow volumes is more suitable to our application than the shadow buffer[RSC87] method due to several factors. These include the limited “field of view” of shadow buffers as well as limited resolution. Finally, the most serious drawback for a dynamic environment is that movement of one object requires recalculation of all of the shadow buffer images.

With the shadow buffer method, projective textures are used to cast shadows in the environment. This is accomplished by creating a light-view depth texture of the environment and using this image to map the shadow. Yet as this light-view analogy implies, this shadow buffer is directed along a particular view and therefore
is limited by the field of view from the light source. To produce the effects of an omni-directional light source, multiple depth maps from several views must be used. This requires multiple renderings of the scene; one rendering must be performed for each view.

In addition to this multiple view requirement, shadow resolution is limited by the resolution of the shadow buffer image for each view. A shadow buffer pixel is mapped into the camera-space image to determine the shadowed area in that scene. Therefore, a single shadowed pixel may be mapped to a large number of image pixels depending on relative image ratios, location of the light source to the objects, and location of the camera. Even with texture sample-filtering functions, this can often result in block pixelation of the shadow.

Finally, the most serious limitation of the shadow buffer method is that since shadows are based on light-view images, object shadows are not independent. Movement of one object requires recalculation of all of the shadow buffer images. This is not conducive to a dynamic environment.

Where the shadow buffer method suffers from the above limitations, the shadow volume method obviates them. The shadow volume is an omni-directional method casting shadows in every direction from a light source. Shadow resolution is at view-space resolution resulting in pixel (or subpixel) accuracy. Object shadows are (mostly) independent\(^1\) which permits recalculation of shadows from only objects which have moved.

The shadow volume technique does however suffer from several shortcomings of its own. These include its reliance on 2-manifold brep objects and uncomplicated polygonal geometry for accurate determination of the silhouette volume and its inability to correctly render shadows when the camera is located inside the shadow volume itself. Both of these problems were addressed within the scope of this work.

---

\(^1\)The dependent case of refractive shadows is stated in the following section.
4.1.1 Silhouette Volume

As described previously, the shadow volume method casts a volume from an object in the direction of the light rays. This volume is drawn for the silhouette edges of the object as seen from the light source casting the shadow. These edges represent the boundary between light and dark areas of the object, and the silhouette volume generated thereby encompasses all areas in shadow. This method therefore requires both the ability to determine which edges of the object are silhouette edges and the ability to assign a direction from the corresponding silhouette face for that edge. The latter is necessary for determining the plus-minus count of the shadowed area.

Determining the silhouette edges of a 2-manifold convex boundary representation (b-rep) requires a simple inner product check of face normals to light rays. Each of the two faces sharing an edge have their normals dotted with the incoming light ray. A silhouette edge is found when these inner products are of opposite sign.

If an object is not a 2-manifold brep, constructing the silhouette volume is not straightforward. An object whose boundaries are 2-manifolds has boundary points with neighborhoods of topological disks. Therefore, any point lying on an edge can be examined as a silhouette edge by examining the direction of a single topological disk. In essence, 2-manifolds guarantee that every edge is a true boundary edge which can be therefore be tested as a silhouette edge as described above. If an object is not a 2-manifold, an edge can lie in more than one topological disk and therefore be shared by more than two faces. Considering that the silhouette face gets its normal direction from the normal direction of the shared faces, having more than two shared faces can present a conflict.

For objects which are not 2-manifold breps, rendering the silhouette volume for every face of the object produces a correct but more costly shadow volume. This method is also necessary for certain instances where the orientation of a face of a 2-manifold brep may be directly in-line with the shadow ray and therefore the inner-product test may fail.
4.1.2 View in Volume

Inherent in the shadow volume methodology is the assumption that the camera is not located in the shadow volume, because a positive count (as opposed to zero) represents a shadowed region. In interior environments, the light is often obscured by some geometry, such as a lamp shade. If the camera is inside the shadow volume, the shadow test must be modified. In essence, we need to cap the shadow volume before the camera.

If the entire camera viewport image plane lay within the volume, capping could simply be accomplished by shifting the shadow volume pixel values by one for the entire scene. Unfortunately, the shadow volume often straddles the viewport, with part of the viewport lying within the volume and part outside the volume as seen in Figure 4.1. In this figure, most of the rightmost cube is incorrectly shadowed.

An actual cap of the volume at the front clipping-plane is most appropriate; in practice, however, it is difficult and expensive to compute. This requires performing the same perspective transform and clipping which the hardware already performs; therefore determining the cap face is a redundant imaging process and is not assisted.
by the rendering pipeline. In addition, this process would introduce other errors in
that this face could not lie on the clipping plane as it would be also clipped away.
An offset would have to be added, which could produce incorrect shading in close
objects.

While calculating a capping face is expensive and introduces clipping errors,
there is a method which uses the shadow volume rendering itself to effectively cap
the volume at the clipping surface. To perform this, note that the cap only need take
place in image space, not in 3-D coordinate space. The cap must simply overlay the
pixel area enclosed by the silhouette volume. Based on this, we note that the image-
space area of the cap of a clipped enclosed volume is equal to the visible internal
pixels of the volume. This observation was noted in the capping of clipped surfaces
demonstrated by Akeley[Ake93b].

While the shadow volume is not an enclosed volume, it is simple to render it as
a single open-ended volume whose opening is at the clipping plane. Normally, the
shadow volume for a single face (bounded plane) with the light source positioned
directly behind it is an open-ended truncated conical polyhedron, with openings at
the top and base. The top opening is bounded by the opaque face itself, with the
bottom bounded by the front clipping plane. If the shadow volume faces are extended
to the light source, the volume becomes an open-ended pyramid whose sole opening
at the base is defined by the clipping plane.

Using this rendering method in conjunction with the cap area relationship, the
shadow volume cap can be readily generated for view-enclosed volumes by performing
a modified second rendering of the affected shadow volume. This second rendering is
performed replacing the shadow volume vertices along the object with the light source
location. In addition, this rendering is performed with Z-buffering disabled, as the
cap is not affected by other geometry since it is located at the front clipping plane.
This rendering also uses the plus-minus stencil function, however the incrementing
and decrementing functions are reversed to thereby cap the internally-visible area.
The use of this method is valid for any shadow volume geometry from any viewpoint. This method does not suffer from any potential “gapping” problems which could occur in actually generating a cap. In addition, this method does not cause false shadows if the view actually lies outside of the volume; the shadow volume remains unchanged. While this fact makes correct shadow generation possible by double-rendering every shadow volume, this method is impractical as it is twice as costly. Instead we perform an intersection check of the light-to-view vector with the bounding-box of the generating object, and subsequently an object-intersection check if the bounding-box test passes. If an intersection occurs, a magnitude check of the view-to-light and view-to-object vectors is performed to validate the view is indeed inside the volume and not on the other side of the light source. Only then is the second rendering of the shadow volume performed.

4.2 Specular Shadows

While the specular image methods described in Chapter 3 do produce accurate reflective images and close approximations for refractive surfaces, they do not produce accurate lighting affects from these surfaces. Light reflects off a mirror and refracts through glass producing different shadows than if not present. To produce a more accurate image, these effects must also be taken into account. Therefore, any shadow generation method must not only work in cooperation with the stenciling method described above, but it must also be affected by the reflective and refractive surfaces in a scene.

To understand how the shadow volume method must be extended for refractive surfaces, examine Figure 4.2. This figure displays the complex shadow patterns caused by objects on both sides of a refracting surface. Note that this is not an exact representation but instead a hybrid model used in our system to greater demonstrate the refracting effects. The rays are refracted as in a change of medium; they do not represent true in-out refraction of a material with a thickness. With in-out refraction,
the refracted rays are parallel to the incident rays and merely offset, thereby not permitting direct light to fall within the light volume [KG79]. Although our included images were generated with this change-of-medium model, in-out refractions are achieved merely by changing the refracting function (or by placing back-to-back refracting faces with opposing indices of refraction in the current model).

To accurately model shadows, each of the above mentioned features must be included in our shadow model. To accomplish this, we require a multi-pass rendering method to produce the effects demonstrated in Figure 4.3.

4.2.1 Intersection Recursion

Our first implementation of specular shadow generation, introduced in [DB94], was a two-pass approach. The first phase generated all shadows and lighting falling within the refracted light area. The second pass renders all lighting and shadows outside this area. This method creates both the shadow and caustic effects of the refractive
Figure 4.3: Refracted Caustics and Shadows

In the first pass, a light volume [Nis87] was generated for the refracting face. Shadow volumes were then generated for shadows falling inside this volume. This itself included two cases: namely objects inside the volume generating shadows and objects outside the volume whose shadows get refracted into the volume. In the first case, the shadow volume cannot intersect the refracting plane for to do so would place the object outside the light volume. In the second case, the shadow volume must intersect the refracting plane in order to be refracted into the light volume. Because true in-out refraction results in refracted rays parallel to incident rays, objects outside the light volume cannot cast shadows into the light volume directly from the primary light source. Both intersection cases can be checked during the shadow volume generation. A simple pre-shadow generation check using dot-products can determine if the object is on the appropriate side of the refracting plane and can save having to generate the shadow volume.

The case of reflection was similar, and the above holds true for single shadow
levels (umbra only). Reflection proved more complicated, however, due to the light volume’s ability to cast shadows into the light volume of the primary light source. With this, objects both outside and inside the reflected light volume can cast reflected shadows. Additionally, objects outside the reflected light volume can cast shadows into the volume. This resulted in several levels of blending from one object, as seen in Figure 4.4.

The second pass of the Intersection Recursion method created shadows for the entire environment. Even the refracted light volume region is included. This captured the shadow effect caused by the refractive surface itself.

4.2.2 Virtual Position Recursion

The original Intersection Recursion implementation of this work suffered from several shortcomings, most importantly were unneeded shadow volume recomputation and incorrect multi-source blending. These issues were addressed in the current
implementation.

As noted in Section 4.2, light rays and therefore shadow rays are refracted or reflected upon interaction with a specular surface. In the implementation described in Section 4.2.1, as shadow rays were traced from the generating object, they were checked for intersection with specular objects. If an intersection occurred, the ray was bent appropriately and the intersection check recursively proceeded for the bent ray.

Unlike the refraction approximation for the secondary viewpoint from Section 3.2, each ray in the Intersection Recursion method was independently bent according to Snell’s Law. This method therefore produced exact shadow tracing for both reflective and refractive surfaces. It can, however, be an expensive computation since intersection checks must occur for each original as well as refracted/reflected shadow ray. If the environment has many specular objects, the intersection checks can be prohibitive for an interactive system. There exist many ray tracing based methods to facilitate this intersection check, but most require some preprocessing subdivision of the environment [FTI86]. In our dynamic environment, these methods are inappropriate.

As the intersection testing is very expensive, an approximate solution analogous to the secondary viewpoint method now replaces the Intersection Recursion method in the current system. This method, Virtual Position Recursion or VPR, requires no intersection checks, and performs recomputation of shadow volumes only when necessary. This method is exact for reflective surfaces, and as accurate as the specular image generation for refractive surfaces; refracted shadow rays use the same paraxial approximation previously described.

Whereas the Intersection Recursion method traced shadow rays to specular objects, the current VPR method computes shadow rays for all refracted and reflected virtual light sources. A reflected shadow ray traces the same path as a shadow ray produced from the associated light source’s reflected position. For refractive surfaces, this virtual light source position approximates the refracted shadow ray direction for
paraxial rays. Therefore, instead of recursively performing intersection checks for each shadow ray, an approximate solution can be achieved by generating shadow rays for all virtual reflected/refracted light positions. In addition, since each ray is associated with a light source and a specular face, these rays can be stored with the generating object and need be regenerated only when either the object, the specular face, or the light source has moved.

In the actual VPR implementation, shadow (silhouette) faces are stored rather than individual rays. Each silhouette face is stored as four vertices and a face normal. All silhouette faces for an object are stored together along with information on the casting object, the generating light source, the recursive level, and the specular object face creating the virtual light position. An update flag is also maintained which is set when one of the shadow-associated objects is moved. Only shadow volumes whose flag has been set are recalculated during the appropriate image pass. While this storage does significantly speed the rendering loop, it does require significant memory overhead. For our bathroom environment with four light sources at one level of recursion and two specular faces, this required 53 megabytes of shadow storage.

During each image pass, which includes the camera view as well as any reflected or refracted views, shadow volumes are stenciled for the entire environment. In addition, this process is recursively called for each specular face in the environment. This generates shadows and incrementally adds lighting from secondary sources. While this method is exact for secondary sources, tertiary and higher order sources are mere approximations unless additional rendering passes are made as the light volumes become much more complex.

As noted above, the current implementation now stores shadow volumes to save recomputation at each frame. While this method significantly reduces rendering times, it requires significant amounts of memory for multiple levels of transport. This is due to the exponential nature of shadow generation when specular transport is involved. To understand this, examine Figure 4.5 in which two specular surfaces are involved. To accurately render the lighting effects emitting from mirror $R_2$ at
two levels of recursion, it can be seen that object $O$ produces two shadow volumes involving specular transport from mirror $R_1$ only, and three shadow volumes involving specular transport from mirror $R_1$ to mirror $R_2$. Each additional specular surface or level of recursion requires an exponential increase because it introduces shadows not only from its final virtual position, but from each of its intermediary positions at each level of recursion. In essence, this is due to the fact that a single object can block the light both reaching and coming from a mirror at each specular transport. The resulting shadows for that object at that level of recursion is the intersection of each of these intermediate shadow volumes.

**Shadow Volume Reclassification**

While it first seems that all of the real and virtual position shadow volumes must be independently stored (and in fact originally were), this is not necessary and indeed
proved too costly. Instead, it can be noted that shadows which pass through a specular surface are simply a transformed version of the original shadow, and therefore a transformed version of the original shadow volume is valid. Again referring to Figure 4.5, it can be seen that shadow volumes $S_{O12}$ and $S_{1O2}$ are reflected versions of $S_{O1}$ and $S_{1O}$ respectively, the former itself a reflected version of the shadow directly from the light $S_O$. Only $S_{1O}$ and $S_{1O2}$ are distinct shadow volumes in addition to the original $S_O$.

This presents a basis on which to re-use, or reclassify, previously-computed shadow volumes. For each virtual light resulting from a specular surface, shadows are generated from an object’s real position (i.e. $S_{1O}$) and from its virtual position (i.e. $S_{O1}$). These virtual-position shadows are transformed versions of the original shadows, as both the object and light have been transformed by the same operation. Therefore, the original shadow volume can be transformed by this same reflection or refraction. This saves recomputation of these shadow volumes, and only one additional matrix multiply of the specular transformation is required before all virtual-position specular shadow volumes are rendered.

Whereas the current implementation generates shadows from objects on both sides of the specular face (shadows which are reflected and shadows from light which is reflected), it also limits its generation of shadows from previous specular transports to two levels. While this method works correctly for interaction of two specular faces, it may add light where it should be obscured in higher orders of specular-specular transport. This problem is somewhat minimized due to the $r^2$ nature of light dispersion.

It must be noted that although the same shadow volume may be used at several recursion levels, view-in-volume checks must be performed at each level based on its transformed position. As mentioned in Section 4.1.2, the view-in-volume check is performed on the bounding box of each segment. To accomplish this, a conventional ray versus axial-aligned box intersection method is used. The light-to-view vector is transformed by the inverse of the transformed global segment’s position, which places
it in the local coordinate system of that segment. This transformed vector can then be checked against the static precomputed local bounding-box of the segment.

In addition, further checks must be performed to omit shadows which were present in a lower level of recursion, but are on the wrong side of one of the successive specular faces along the transport. Otherwise an object behind a mirror could cast a false shadow after its volume is transformed. This is accomplished with checks of the shadow volume against the view and light position relative to the specular surface. While this must be performed for each recursion level as the shadow volume is transformed, the check is readily accomplished with a simple inner-product test of the transformed positions. The simplicity and inexactness of this view-side check introduces further complexities, however.

As is readily apparent, specular shadows should be only those shadows which fall in the light volume. Exclusion of shadows based solely on which side of the specular face they originate fails to eliminate many shadow volumes which fall outside this light volume. These “false” shadow volumes currently also get rendered, although they affect only objects outside the stenciled light volume. While it first seems it would not cause a problem as only objects in the light volume (stenciled area) are re-rendered, it can for certain views create erroneous shadows. This is due to the reclassification of shadow volumes and the nature in which they are generated.

When shadow volumes are generated, they are assumed to be “capped” at one end by the generating object itself. This assumption is no longer valid when these shadow volumes are reclassified (and moved) for specular shadows. The generating object is no longer rendered, and therefore the transformed shadow volume is now open-ended. For certain environments where shadow volumes are rendered outside the light volume, the view may be looking into the shadow volume from where the generation object would normally be. This presents a situation analogous to the view-in-volume problem addressed previously; subsequently an analogous and even simpler solution is available.

As the reclassified specular shadow volumes now originate on the light side of
the specular surface, enclosure of objects before the specular surface can be ignored as those pixels are not re-rendered. Therefore, we can make a closure of the open end by again extending the vertices which originated at the silhouette edges up to the light source. This extension prevents viewing the inside of the volume through the opening where the originating object would normally be. This creates an open-ended pyramid which can then be processed as normal including full view-in-volume checks.

4.3 Light Volumes

When light interacts with a specular surface, some of the light is reflected or transmitted, thereby creating an enclosing volume of the specular light. This light volume can be viewed as light coming from a secondary source whose position is the virtual position of the light source reflected or refracted by the specular surface and whose intensity is modulated by the specular material. This light source reclassification [CRMT91] in conjunction with an associated light volume is the basis for specular illumination in our system.

4.3.1 Single-Bounce Light Volumes

Light volumes from secondary sources are generated in the same Virtual Position Recursion manner as the shadow volumes. Just as shadow volumes stencil the area in shadow, light volumes stencil the area in which light from secondary sources can be added, and subsequently omitted thereby causing shadows. The light volume is generated using the same plus-minus shadow volume method, however the stencil values are in essence inverted to permit rendering only in the “shadowed” area. Whereas with shadows the zero area represents the area not in shadow and therefore the valid rendering area, with light volumes the zero area is the area outside the light volume. We therefore set all pixels of shadow-zero to be the refractive-zero value (and therefore not valid in subsequent shadow calculations), and conversely
all non-zero stencil values which are greater than the shadow minimum to be the shadow zero value. Just as the shadow-zero value determines the valid render area for rendering shadows in a mirrored image, it also determines the valid area within a light volume. This refractive use of the shadow zero value is further detailed in Chapter 5.

4.3.2 Multiple-Bounce Light Volumes

The original VPR implementation of this method, which eliminated the intersection check of shadow and light volumes with specular faces, generated the light volumes based solely on the virtual secondary light source position and the current specular face. This is not accurate, however, with sources involved in multiple specular-specular transports. With each specular-specular transport [WCG87], the light volume is bent by each specular face through which it passes, and therefore the corresponding virtual light position is transformed as well. Furthermore, the light volume is reduced by each of the specular faces. This process is seen in Figures 4.6 and 4.7. This first implementation correctly calculated the virtual light position, but the light volume was only effectively “clipped” by the final specular face. In cases where the visibility of the mirrors to each other is obscured or where only a sliver of light is produced, this may cause lighting errors. These errors are in many cases minimal considering the $r^2$ fall-off of light sources as well as non-perfect reflectance of materials.

While in many cases the described light-volume rendering is fairly accurate, it is not an exact solution. There are, however, methods to overcome this limitation which were not fully implemented within the scope of this work due to their rendering costs but are briefly discussed for completeness. These methods are based on the same principles as described above, but involve further reduction of the light volume and additional rendering of shadow volumes.

As demonstrated in Figure 4.6, a light volume from a tertiary source should be
clipped by both specular faces through which it passes. This includes the specular face $R2$ reflecting the image of a specular face $R1$ reflecting the light as well as the face directly reflecting the light. This can be accomplished by either direct reduction of the light volume through intersection checks, or by multiple renderings of the light volume through each face to determine the valid overlapping region.

The first method is the basis for the earlier Intersection Recursion work, whereby shadows and light volumes were checked for intersection with specular faces. Where intersections were found, recursive processing of the spawned volumes proceeded. This method, while more exact, suffers to some degree from the same performance restrictions which ray tracers face.

The second method is based on additional rendering passes of the VPR implementation. As the correct light volume is the light which passes through each specular face along its path, this volume is also the intersection of the light volumes generated through each specular face. Therefore, the correct light volume can be created by stenciling each of the constituent light volumes to determine the overlapping area. This is the method which is implemented in the current system for two levels of specular transport, which is seen Figures 4.8 and 4.9 for reflective and refractive surfaces at a recursive depths of one and two. Figure 4.8 shows the generated light

Figure 4.6: Light volume clipped by both refracting planes.

Light Volume refracted by $R1$ & $R2$

Refracting Planes

R2

R1

Lightsource

Virtual Lights

R2($R1(L)$)
volumes for the corresponding shadowed environments in Figure 4.9. Whereas this method is exact for reflective surfaces, as demonstrated in Figure 4.7, it is an approximation for refractive surfaces. Again referring to Figure 4.6, it is apparent that the intersection of disjoint light volumes fails to account for instances where only a partial light volume intersects a refracting face.

In addition, the current VPR implementation only uses two levels of light volume clipping. While higher orders of specular transport are possible in the current system, the light volume clipping itself is limited to the latter two specular faces. This approximation is not an inherent limitation of the methodology, rather it was an implementation decision based on the goal of an interactive rendering system.

4.4 Light Accumulation

While the above implementation addressed the need to recompute shadow volumes at every frame, its benefit is limited without proper blending of the resulting shadow effects. The original implementation of this work relied on use of the accumulation buffer to overlay the independent shadow effects from real and virtual light sources. This not only produced incorrect blending as ambient light was multiply added, but
Figure 4.8: VPR Light Volumes.
(a) 2 reflective surfaces, depth 1  (b) 1 reflective/refractive surface, depth 1

(c) 2 reflective surfaces, depth 2  (d) 1 reflective/refractive surface, depth 2

Figure 4.9: VPR Lighting and Shadows.
it also precluded the use of the accumulation buffer for other purposes such as light jittering or antialiasing.

The current implementation performs the compositing of independent light contributions without using the accumulation buffer. Instead, the hardware blending function is used in coordination with successive rendering passes which rely on visible surface computations from the initial rendering pass. The accumulation buffer is then used with jittered lights to accumulate complete images and produce soft shadows. These methods of light blending and accumulation are fully detailed in Section 6.1.

In addition to proper blending, the lighting contributions from the virtual light sources must themselves be modulated from the original source. This modulation occurs due to the partial transmittance or reflectance of the specular material as well as other surface properties of the material. We include two means of modulation of the reclassified virtual light; these are through attenuation and filtering.

### 4.4.1 Attenuation

As no surface is purely specular, not all light impinging on a surface is specularly transmitted or reflected. The amount of light which is transmitted or reflected is attenuated by the material properties of the surface.

For pipeline rendering, each specular surface and lightsource has an associated light volume which demarcates the specularly illuminated area. This area is illuminated during a successive rendering pass by a lightsource at its virtual position. This lightsource’s emittance is first attenuated by the specular properties of the reflecting or refracting surface. Thus, the light is effectively “dimmed” by each specular surface with which it interacts. In addition, further attenuation occurs simply because of the virtual position of the lightsource. This virtual position, along with the attenuated color, are used in the hardware lighting equation.
4.4.2 Filtering

In the previous sections, light volumes were created for light passing through transparent glass or reflecting off mirrored surfaces. These methods assumed specular surfaces of uniform coloration, density, and transparency. There are, however, many common materials which do not fall into this category. This includes beveled glass, prisms, and stained glass. Light interaction with these materials can, however, be simulated using a technique common to rendering complex materials, namely texture mapping.

As light passes through a surface, it may be filtered through the surface thereby projecting a distorted image of the specular surface onto some diffuse surface. This projection of an image is the same principle used in projective texture mapping.

Projective textures have been previously used for a variety of purposes, such as a slide projector onto a surface [DSG91], and as a shadow caster based on a depth texture map [Wil78]. In addition, this method has been used to apply a corrective perspective transformation to an image produced from a given vantage point [KNN+89], and conversely, Dorsey [Dor93] used this principle to produce the perspective distorted image itself.

While the principle of projective textures has appeared in computer graphics for years, it is only recently that many modern graphics architectures support some method of texture projection in hardware. Segal et al. [SKvW+92] demonstrated how projective coordinate transformations permit texture coordinate assignment based on the depth maps described in [RSC87]. With this facility available, we can simulate light interaction with many of the above mentioned materials.

The principle of projective texturing relies on transformation of world coordinates to clip coordinates as well as of world coordinates to light coordinates. In each of these cases, the transformed 3-D coordinates are mapped into a 2-D local coordinate frame of the light (projector) and the eye. These 2-D coordinates are given by \( x_s = x/w \) and \( y_s = y/w \) for the screen coordinates and \( x_l = x_l/w \) and \( y_l = y_l/w \)
for the texture coordinates. The corresponding transformation relating the texture coordinates to screen coordinates, as given in [SKvW+92], is

\[ Q' = \frac{aQ'_1/w_1 + bQ'_2/w_2}{a(w'_1/w_1) + b(xw'_2/w_2)}, \]

where \( Q' \) is the texture coordinate corresponding to a linearly interpolated point along a line segment in screen coordinates.

As seen in Figure 4.10(a), this transformation is dependent on local axial-aligned 2-D coordinate frames. Unfortunately, this is seldom the case with arbitrary environments where the texture plane (i.e., the specular surface) is not axial aligned with the light view. This is demonstrated in Figure 4.10(b).

As can be seen in Figure 4.10(c), there is a corresponding projection \( M_{33} \) of this texture plane onto the axial aligned texture coordinates. As with the image mapping of a refractive image onto a specular plane described in Section 3.2, this transformation is a 2-D projection of quadrilateral to quadrilateral. In fact, this instance is the simpler mapping of rectangle to quadrilateral, described in [Hec89] and included in Appendix A.

The application of this transformation is seen in Figure 4.11, where the first image shows normal hardware-based texture projection and the second shows the same projection with a non-axial aligned texture plane. As with the refraction transformation, this requires a 2-D projective transform which is simulated with a corresponding 3-D transform. Note that this approximation also affects Z-buffering, which in turn affects the visible surface mapping in the environment. A more flexible pipeline architecture would enable true and correct implementation of arbitrarily oriented texture plane projections.

While the above implementation is suitable for general slideshow projection, it alone is not suitable for simulation of light passing through a specular surface. This requires only projection of the texture on surfaces receiving light. In the previously described shadow implementation, all contributions of both real and virtual light sources occur after all shadow volumes of that source have been stenciled. This
Figure 4.10: Light and view coordinate systems.
provides a convenient rendering stage for the projected texture, namely the rendering of the light volume. By applying the texture only during this stage, the light volume is thereby modulated by the texture. This produces the desired effect, which includes projection of the material properties only in areas not obscured by shadow. A detail of these effects is seen in Figure 4.12, where four real lights and four virtual sources shining through a “beveled” glass door produce eight partially overlapping texture patterns which are occluded by the door handle shadows.
Figure 4.12: Projected Texture Light Pattern
Chapter 5

Multi-pass Process

In the previous chapters, the individual components of a multi-pass rendering system were introduced. This includes shadow generation for real and virtual light sources and specular image generation from a virtual viewpoint. Both features rely on manipulating the hardware matrix stack to create these virtual positions. Both features require multiple passes to render for these virtual positions. Additionally, both features use stenciling to mask to appropriate screen regions for each rendering pass. This chapter examines the interaction of these two similar rendering processes.

5.1 Recursion

The coordination of the separate processes is seen in the following pseudo-code.

draw_window(Camera)
{
    if (SPEC_ON) //if reflection/refraction enabled
        draw_spec_objects(Camera); // -draw reflected/refracted view
    if (SHADOWS_ON) //if shadows are enabled then...
        turn_lights_off(); // -turn all lights off
        draw_objects(); // -draw nonspecular objects unlit
        for (each light) // -loop over each light and...
            make_shadows(); // -draw shadows in light volume areas
    else //else if shadows aren't enabled then...
        draw_objects(); // -draw nonspecular objects lit
}
draw_spec_objects(Camera)
{
  for (each spec_face)
  {
    mask_face(spec_face); // set stencil area for face
    reclassify_camera(Camera); // move camera to virtual viewpoint
    draw_window(Camera); // recursively draw reflected/refracted view
  }
}

make_shadows(light)
{
  env_sten(AFT_SPEC); // objects inside light volume (after light has reflected)
  if (shadow_level>0) // if shadowing a light volume
    env_sten(PRE_SPEC); // objects outside light volume (before light has reflected)
  turn_light_on(light); draw_objects(); // add light to scene (except where shadow volumes)
  turn_light_off(light); make_spec_shadows(shadow_level++); // add light from virtual sources (reflected/refracted light)
}

make_spec_shadows(light)
{
  for (each specular_face)
  {
    spec_light(light,spec_face); // move light to virtual reflected/refracted position
    make_light_volume(light); // mask area where light volume falls
    make_shadows; // create shadows in light volume area
  }
}

This process is readily apparent in Figure 5.1, which demonstrates the rendering sequence for an image with two specular faces. The entire scene is rendered for each specular face from the virtual viewpoint of that face, with the face attribute then blended with the final specular image. This represents the compounded virtual position if several levels of specular transport are involved. For each scene rendering, multiple passes are required for complete lighting and shadow generation. This process is detailed in Chapter 4.

5.2 Stenciling

Both the rendering of shadows and the rendering of specular images require use of the stencil planes. For specular image generation, the stencil plane not only masks the valid rendering area, but also acts as a recursive counter to enable depth-first traversal in the recursive rendering process.

We choose zero for our render area stencil value; this is the stencil mask value
Figure 5.1: Recursive Image Rendering
for drawing at every level of recursion. At each level of specular image recursion, all stencil values are incremented by one (setting the previous rendering area to one), and the new specular surface is drawn setting the stencil value to zero. This creates a mask of zero stencil values at the pixels where the specular surface was rendered (the visible portions). The scene from the refracted or reflected virtual viewpoint is then drawn in this zero stencil area, and the process is repeated for all other specular surfaces. Once the desired recursion level has been reached, all stencil values are decremented (with zero capped), which essentially pops back one level of recursion. The process is then repeated for the next recursive surface, with stencil values incremented by one and the surface creating a stencil mask of zero. This process is illustrated in Figure 5.2.

Figure 5.2: Recursive Stenciling

At each level of recursion, shadows must be drawn in the valid area. This area may include previously rendered specular surfaces so that shadows may be cast on
partially specular objects. These surfaces are rendered first as purely specular and then blended with their respective material properties. As described above, specular surfaces have their stencil values reset to zero after their specular image has been rendered. The valid rendering area is therefore popped back to zero, with lower recursive depths maintaining higher stencil values.

While it might seem that the zero specular stencil mask value would be a logical choice for the zero value in the plus-minus shadow algorithm, this is not the case. In order to have recursive reflections and refractions, we instead choose a value which is three-fourths of the maximum stencil value for our “zero” shadow value (SHAD_ZERO), and one-half of the maximum for our minimum shadow stencil value (SHAD_MIN). This provides half of the stencil buffer for shadow calculation and half for recursive specular levels. These values can be adjusted according to the recursion level needed or the shadow object complexity.

The reason for our choice of SHAD_ZERO is now apparent; it avoids conflict with our recursive refraction levels. All stencil values of zero at each level are changed to SHAD_ZERO, and shadows are then rendered as described above using the plus-minus method. SHAD_ZERO should be chosen so that the plus-minus method does not go below the shadow minimum (SHAD_MIN) or above the maximum stencil value in order to prevent conflict with the specular recursion stencils. After all shadows are drawn, all stencil values greater than SHAD_MIN are reset to zero, for continuation of the specular recursion. The basic stenciling procedure is seen in the expanded pseudo-code functions below:

```c
draw_window(Camera)
{
    if (SPEC_ON){
        SPEC_LEVEL++;
        draw_spec_objects(Camera);
        SPEC_LEVEL--;
    }
    if (SHADOWS_ON){
        apply_stencil(EQUAL, ZERO, REPLACE, SHAD_ZERO);
        turn_lights_off();
        draw_objects(); //ambient only
    for (each light){
```
make_shadows();
apply_stencil(GREATER, SHAD_MIN, REPLACE, SHAD_ZERO);
}
apply_stencil(GREATER, SHAD_MIN, REPLACE, ZERO);
else
draw_objects();
}
apply_stencil(COMP_FUNC, COMP_VALUE, PASS_FUNC, PASS_VALUE)
{
    stencil(COMP_FUNC, COMP_VALUE, PASS_FUNC, PASS_VALUE);
    apply_to_screen(); //apply stencil to every pixel
}
/*REF ROUTINES (Refraction and Reflection)*/
draw_spec_objects(Camera)
{
    if (SPEC_LEVEL != 1)
        clear_stencil(ZERO);
    apply_stencil(EQUAL, ZERO, REPLACE, SPEC_LEVEL);
    for (each spec_face)
        stencil(EQUAL, SPEC_LEVEL, REPLACE, ZERO);
        draw_face(spec_face); //SPEC_LEVEL > 0 where face
        VirtualCamera = reclassify_camera(spec_face, Camera);
        draw_window(VirtualCamera); //draws only where stencil == 0
        apply_stencil(EQUAL, ZERO, REPLACE, SPEC_LEVEL);
    apply_stencil(EQUAL, SPEC_LEVEL, REPLACE, ZERO);
    stencil(EQUAL, ZERO, ST_KEEP, ZERO); //draw where 0
}
/*SHADOW ROUTINES*/
make_shadows(light)
{
    env_sten(AFT_SPEC); //objects inside light volume (after light has reflected)
    if (shadow_level > 0) //if shadowing a light volume
        env_sten(BFR_SPEC); //objects outside light volume (before light has reflected)
        turn_light_on(light);
        stencil(EQUAL, SHAD_ZERO, ST_KEEP, SHAD_ZERO);
        draw_objects(); //add light to scene (except where shadow volumes)
        turn_light_off(light);
        make_spec_shadows(shadow_level++); //add light from virtual sources (reflected/refracted light)
}
env_sten(side)
{
    if (side == PRE_SPEC)
        transform_shadow_volumes(); //shadow volumes move to virtual position
    for (all silhouette faces)
        stencil(GR_EQUAL, SHAD_MIN, facing_view(sil_face)?ST_INCR:ST_DECR, 1); //in/out method
        draw_silhouette_face(sil_face); //draw face without color to create stencil mask
    }
}
Chapter 6

Illumination Model

While the preceding chapters presented light and shadow volume methods for real and virtual light sources, the discussion was without regard to overall scene illumination. The illumination of a scene consists of several factors. Illumination can occur either directly from a primary light source or from a secondary virtual source such as the reflection of a light in a mirror or from refraction through glass. Illumination of this type from a real or virtual source is termed direct illumination. Direct illumination which is dependent on scene geometry (as shadows may occlude primary and virtual sources) is termed the global direct illumination. Light which does reach a surface is governed by the material properties of that surface and by the properties of the light source. This is called the local direct illumination. In addition, the surfaces of the environment may reflect light back into the environment, producing indirect illumination. This illumination is global in the sense it is dependent on scene geometry and occlusions. Each of these factors as they relate to a pipeline rendering system are examined in turn.
6.1 Global Direct Illumination

While the recursive shadow algorithm does handle the constituent effects of direct primary and secondary source illumination, it alone does not guarantee correct blending of these effects. When used with traditional pixel overwrite (i.e. no blending), it produces only the umbra of the shadows. Areas in shadow from only some light sources appear as bright as those in no shadow, and only areas in shadow from all light sources appear in total shadow. In order to produce the penumbra effects or areas only in partial shadow, an accumulation of the lighting effects from the primary as well as reflected/refracted lights must occur. Fortunately, we can use an accumulation buffer or blending functions to do just this.

There are two methods for performing this accumulation of lighting effects. The first method is to treat each shadow calculation as independent and sum each resulting image. Areas which receive light from both the source and reflected/refracted light volume produce caustic effects. This method has limitations when using a single accumulation buffer due to the inability to create intermediate images while preserving the final accumulated image, but this can be overcome through the use of hardware pixel blending during the rasterization phase.

The second method uses an extension of shadow volumes by [BB84] for soft shadows. By processing all shadow volumes without producing intermediate images, the stencil value of each pixel represents a “darkness level” due to encasement in several shadow volumes. The actual lighting of the scene is performed after all shadow volumes have been generated. Assuming the darkness level is only greater than the zero stencil value, the scene is redrawn with diffuse lighting for areas whose stencil value is less than the shadow zero value. All stencil values are then decremented, and the image is redrawn. This process is repeated for each darkness level, accumulating each intermediate image. This produces a final image with intensities based on the number of enclosed shadow volumes. This method does not suffer from the problems inherent in the first method; however, extensive stencil value “juggling” of the image
is necessary.

### 6.1.1 Light Accumulation

The VPR implementation described in Chapter 4 uses the above method of accumulating individual lighting contributions by rendering each separately. The original Intersection Recursion implementation was limited in the shadow effects which are possible as it relied on blending of final images using the accumulation buffer, as described in Section 4.4. Refracted shadows from a single light source with caustics and multiple shading levels were supported by the Intersection Recursion method, however reflected light or light from multiple sources results in incorrect blending (see Section 4.2.1). This was due to the limitations of the accumulation buffer blending, which provides only addition to a single image; multiple lights require blending of several final images.

To overcome this shortcoming, blending was removed as a final stage process, and instead individual light contributions are added in RGB space at every iteration based on the principle of linearity of light transport [Dor93]. The first-pass rendering is performed with ambient light only; successive iterations are performed with no ambient light. The resulting light contributions are added to the composite image through use of the blending function. This modification required changing the Z-buffer comparisons of lit rendering passes.

With the original implementation, each rendering pass re-rendered the entire scene using Z value sorting for hidden-surface calculations. While this method is sufficient when blending occurs using final visible-surface images, it is not valid when blending “on the fly”. With Z-buffered sorting, hidden surfaces may be temporarily rendered until later overwritten by a closer surface. Because of this, hidden pixels would be blended in the final image even though they are overwritten later in the rendering loop. To produce a correct image, only visible pixels should be added to the accumulated image, and therefore only visible pixels should be rendered in the
lit stages of the rendering cycle. Luckily this can be accomplished as these pixels are effectively tagged after the first unlit rendering iteration. This enables successive rendering passes to effectively skip Z-buffer sorting.

The initial rendering with only ambient light is performed using normal Z-buffer sorting. This produces not only an unlit rendering of the scene, but also a Z-buffer containing all rendered pixel values. By performing all successive renderings with a Z-buffer comparison function checking for equal values \((ZF_{EQUAL})\), only those visible object pixels are re-rendered and thereby blended with the final image.

This method relies on the assumption that all successive renderings of the same object will generate the exact same Z values as the initial rendering. This implies that the camera position must not move (as in jittering) between iterations. Also, because of modern hardware rasterization, polygons must be rendered in the same vertex order as the original rendering; rendering the same polygon in different vertex order may result in slightly different Z value rasterization. Even with these constraints, certain hardware may still generate different Z values as the rasterization is affected by the lighting mode path. This is a serious hardware limitation when it arises in that to effectively handle it, pixel tagging must be performed by some other method such as the stencil buffer, which is already severely taxed by the other pipeline rendering components.

### 6.1.2 Soft Shadows

While the accumulation buffer proved unworthy for generating partial shadowing effects due to its reliance on blending final images, this functionality is well suited for blending light-jittered images to produce soft shadows.

In the original implementation, the accumulation buffer was used for accumulation of shadow effects. With the replacement of the accumulation buffer with the rasterization blend function, the accumulation buffer was freed to perform final image blending. While the initial use was for anti-aliasing, this same jittered
multi-rendering process was modified to perform soft-shadow blending.

Whereas antialiasing is performed by jittering the camera position in a stochastic manner, soft shadows are created by jittering the light source[BB84]. This in effect approximates an area light source with a stochastic sample of point light sources.

The actual implementation of this source jittering is an approximation to an actual movement of the source point. Recall that shadow volumes are computed once and stored with the corresponding object. For each silhouette edge, a quadrilateral shadow volume face exists. These faces contain no information on the original light position. Therefore, modification of the light position would require recomputing every shadow volume face at each jitter iteration or initial storage of the entire set of jittered faces. Neither of these options is an attractive proposition for an interactive system.

The alternative solution notes that with a jittered source, only two of the shadow volume face’s four vertices are moved, and that their movement is a linear transformation of the stochastic jittering of the source. As the distribution is centered around the point source position, any rotation of this distribution also produces a valid stochastic sample.

Another factor in this approximation is the assumption that the jitter distance is relatively small compared to the distance of the light source to the object. The object effectively acts as a pivot point in the shadow volume calculation, whereby if the object is close a lever-effect produces a large movement in the two moving shadow volume face vertices. Assuming all ratios of jitter distances to light-to-object distances are relatively equal, we can approximate the source jittering with a linear multiple jittering of the two distant vertices.

As can be seen in Figure 6.1, a movement of the light produces a linear multiplied movement of the end of the shadow volume. Figure 6.2 demonstrates the above approximation, along with the error introduced by this method. This approximation can be made exact by including with each silhouette face the distance of the light to each of the two nodes bordering the segment. As the segment acts as a pivot, the
Figure 6.1: Jittered Light Shadow Volumes

Figure 6.2: Jitter Approximation Shadow Volumes
ratios \( b_2/b_1; a_2/a_1 \) relate the displacement of the two generated nodes of the shadow face to the two nodes bordering on the generating face edge itself. In practice, the jitter size as compared to the distances involved is relatively minute, and a constant multiplier usually suffices. The current implementation of this work uses this constant multiplier to reduce memory requirements.

The above jitter approximation therefore provides a simple way to jitter the shadow volume without recomputing it. In addition, the jitter amount can be scaled according to the number of samples being blended through a simple multiplier. The same multiplier can be used to control the area light source, and the shape can be directed by the jitter sample distribution itself.

### 6.2 Local Direct Illumination

In Section 6.1 we demonstrated techniques for overlaying and blending multiple renderings to create more realistic images. This involved the linear combination of independent lighting contributions as well as shadowing to provide global illumination effects. We have not yet considered the local illumination model. The local model represents the surface shading from direct illumination from real and virtual light sources. These lighting contributions are generated using the available graphics hardware shading model, which is generally Gouraud shading using the Phong lighting model\(^1\). In this model, the cosine-based Phong lighting model is used to determine vertex intensities, while Gouraud shading interpolates between neighboring vertices. While the use of the Phong model is widespread due to its simplicity, it is, however, a purely empirical model (and not a particularly good one) and is widely known for its inaccuracies, foremost being its non-reciprocity.

There has been much study devoted to both realistic and physically accurate lighting models. This work is generally centered around the rendering equation

\(^1\)Although we (and the hardware designers themselves) refer to the various implementations of exponent-based models as Phong’s model, many are actually Blinn’s\([Bli77]\) model. Usually mistakenly though of as equivalent, the two models are different albeit closely related\([FW94]\)
introduced by Kajiya[Kaj86]. This equation, expressed in terms of reflectance

\[ L_r(\theta_r, \phi_r) = \int_0^{\pi/2} \int_0^{\pi/2} L_i(\theta_i, \phi_i) \rho_{\theta i}(\theta_i, \phi_i; \theta_r, \phi_r) \cos(\theta_i) \sin(\theta_i) \, d\theta_i \, d\phi_i \]  

(6.1)

states the relationship between incident light and reflected light from a surface based on a bidirectional reflectance distribution function (BRDF)[NRH+77]. The BRDF itself is a function dependent on the two incident and two reflected angles, as seen in Figure 6.3. It is bidirectional in that it depends on both the incident and reflected angles and these directions can be reversed without changing the evaluation of the function. This implies the reciprocity property of light scattering which the Phong model lacks.

Many BRDF formulations have been developed to try to represent this complex, high-order function. This includes creation of theoretical physics-based models [HTSG91] [TS67] [CT82] and the use of spherical harmonics [CMS87] [SAWG91] [KV84]. More recently, these methods have been extended for anisotropic surfaces [War92] [WAT92].
While these advances have continued to improve ray-traced and radiosity images, hardware rendering quality has been relatively static with its basis on the Phong model. The hardware Phong model is not a true BRDF:

\[
C_p = C_{me} + C_{sa} C_{ma} + \sum_i \frac{C_i}{K_{sdef} + K_{sdev} D_y} \left( C_{md} \left( \hat{N}_{pl} \cdot \hat{N}_p \right) + C_{ms} \left( \hat{N}_b \cdot \hat{N}_p \right) E_{mss} \right)
\]

(6.2)

where the \( C \) values are the scene, material, and light colors, the \( \hat{N} \) vectors represent the normal, bisector vector, and direction to light as seen in Figure 6.4, the \( K \) values are fixed and variable attenuation factors, and \( E_{mss} \) is the glossiness exponent. In fact, the Phong model does not guarantee conservation of energy as the specular term actually acts like a second diffuse term for low glossiness terms.

![Figure 6.4: Phong BRDF Notation](image)

While the traditional Phong representation is not founded in the physical transport of light, it can be made more accurate if fit to a physically-based model. While the more-general anisotropic models are far beyond the capabilities of the Phong
model, isotropic models provide a good reference scattering function close to the range of the Phong model. In particular, we used the isotropic Gaussian model presented by Ward [War92] to perform a Chi-Square fit of the Phong lighting model. This model provides a metric against real-world illumination in that it incorporates actual measured material parameters and has itself been tested against gathered data. The specular component of the Gaussian BRDF is

\[
\rho_s \frac{e^{-\frac{\tan(\delta)^2}{\alpha^2}}}{\sqrt{\cos(\theta_i) \cos(\theta_r) |4\pi \alpha^2|}},
\]

and the corresponding Phong specular term rewritten from Equation 6.2 is

\[
\rho_s \cos(\delta) E_{mss}
\]

where \( \delta \) is the polar angle between the half-vector and the surface normal as seen in Figure 6.4.

![Gaussian Specular](image1)

![Phong Specular](image2)

(a) Gaussian (\( \alpha=0.08 \))  
(b) Phong (\( E_{mss}=318 \))

Figure 6.5: Specular Scattering Components (\( \theta_i = \pi/6 \))

As can be seen with comparison of the specular components of the Phong versus the Gaussian model in Figure 6.5, both functions peak where the incident and reflection directions are opposite (although off-peak phenomena as detailed in the Torrance-Sparrow model is not demonstrated). The Gaussian model uses a roughness term \( \alpha \) to modulate the concentration of the specular reflection, while the Phong model uses a glossiness exponent \( E_{mss} \) to accomplish the same. A low roughness value
is analogous to a high glossiness term, with 0 and \( \infty \) the respective values for a pure specular mirror. But while the Gaussian model falls off to zero\(^2\) as \( \alpha \) increases, the Phong model actually approaches one as \( E_{\text{mss}} \) decreases (see Figure 6.6).

By examining the Phong specular term, it can be seen that the glossiness is not the only parameter which determines the specular component. The specular reflectance \( \rho_s \) also modulates the specular component, and through modification of the reflectance based on the glossiness, the Phong reflectance term can approximate the isotropic Gaussian model. By performing a two-parameter Chi-Square fit of the Phong specular term using a parameterized glossiness value as well as a linear scale of the glossiness value as a specular reflectance multiplier in

\[
P_1 P_2 \rho_s \cos(\delta)^{P_1},
\]

we can achieve an almost exact approximation (i.e. \( \chi^2=0.00000037 \) @ \( \alpha=.08 \)) at any given \( \alpha \) of the following modified Gaussian specular term

\[
\rho_s e^{-\left(\frac{\tan(\delta)}{\alpha^2}\right)^2}\frac{1}{4 \pi \alpha^2}.
\]

This simplified term omits the parameters unavailable in the Phong representation as seen in Figure 6.7. Even with the full Gaussian specular term, a two-parameter

\(^2\)This fall-off to zero is actually incorrect behavior. Ideally, the Gaussian model should fall off to \( \rho_s/\pi \). For the purposes of our approximation where roughnesses are under 0.2, a zero limit suffices.
Chi-Square fit ($\chi^2 = 0.02246598$) can be performed using the following model

$$P_2 \left( P_1 \frac{1}{\alpha^2} \right) \rho_s \cos(\delta)(P_1 \frac{1}{\epsilon^2})$$  

(6.7)

with the glossiness parameter replaced with a second parameter which additionally relates the Phong glossiness term to the Gaussian roughness. This is demonstrated for several roughness terms in Figure 6.8.

![Simplified Gaussian vs. Fit Phong](image)

(a) Wireframe    (b) Contours

Figure 6.7: Simplified Gaussian vs. Fit Phong

It can be seen by this comparison that the models diverge as the angles become more obtuse. This behavior and inadequacy of the Phong model was vividly noted by Blinn [Bli77].

While the Phong model can be fit to the Gaussian BRDF, this fit model is not applicable to the hardware lighting model. If we examine the full diffuse and specular components of the hardware model, we note that $\cos(\theta_i)$ is used to modulate the light received by the angle incident to the surface normal by the source. The rendering equation (eq. 6.1) similarly uses a differential solid angle to represent the projected solid angle subtended by the source. This introduces an additional $\sin(\theta_i)$ term not included in the hardware equation. While this term integrates out for a true point light source, it is present in Monte Carlo sampling systems such as Radiance [War94] where there are no true point sources. Therefore, to provide some consistency between the diffuse and specular computations as related to a
Figure 6.8: Full Gaussian vs. Fit Phong
physically-based system, the \( \sin(\theta_i) \) term is dropped from the rendering equation and the Phong model is fit to the rendering equation. The specular term therefore becomes

\[
\cos(\theta_i) \rho_s \frac{e^{-\frac{\cos(\theta_i)}{\sigma^2}}}{\sqrt{\cos(\theta_i) \cos(\theta_r) |4\pi \sigma^2}}
\]

(6.8)

A Chi-Square two-parameter fit of this model produces the following Phong term:

\[
.0398 \left( \frac{1.999}{\alpha^2} \right) \rho_s \cos(\delta) \left( \frac{1.999}{\sigma^2} \right)
\]

(6.9)

with a corresponding \( \chi^2 \) value of 0.00008003. The exactness of this fit is seen graphically for several \( \theta_i \) in Figure 6.9.

![Figure 6.9: Partial Gaussian vs. Fit Phong](image)

This model provides a much better physical approximation than the un-normalized Phong model, as seen in Figures 6.10 and 6.11. As can be seen in the figures,
the intensity is modulated according to $\alpha$ in both the Gaussian and Fit Phong, yet remains constant in the standard Phong model. Note, however, that the hardware Phong model is limited in its accepted value for $E_{\text{max}}$; values are capped at 128. This is seen in overly-broad specular highlights for $\alpha = .03, .07, \text{and } .10$. Even with the Phong model limitations, a true implementation permitting a full range of exponents would provide a better illumination model than is currently feasible.

![Gaussian vs. Phong vs. Fit Phong](image)

Figure 6.10: Full Gaussian vs. Phong vs. Fit Phong (view 1)

While the Fit Phong model is founded in a more physical basis, it also does not include the grazing effects of the original Gaussian. It tends to overstate specular highlights for acute $\theta$s, and conversely understates obtuse grazing angles. This behavior is apparent in Figure 6.12, which shows contrasts of the specular highlights of Figure 6.14. Because the Phong model has only a notion of $\delta$ and not the corresponding $\theta$s for specularity, the model itself is severely limited as to the specular
Figure 6.11: Full Gaussian vs. Phong vs. Fit Phong (view 2)
effects it can provide. Even with these limitations, the benefits of the fit model are further magnified by the ability to now incorporate measured material parameters as in [War92], thereby providing physics-based material properties.

Gaussian vs. Overstated Phong (Small $\theta_s$)

Gaussian vs. Understated Phong (Large $\theta_s$)

Figure 6.12: Gaussian vs. Phong Specular Highlights

In addition, the hardware lighting model is also limited in its lighting calculations and display in general. All lighting calculations are generally limited to 8-bits per channel, and even when its unit-less calculations are fit to a physical model as above, display conversion is itself severely limited. We have not attempted to overcome these hardware limitations. Work by [War91] and [TR93] suggest how they could be addressed in an expanded real-value lighting model incorporating tone reproduction in its calculations.
6.3 Indirect Illumination

As mentioned previously, radiosity systems are well suited for calculation of global illumination effects. This includes direct diffuse illumination from light sources as well as indirect illumination reflected from surfaces in the environment. It does not, however, capture the specular illumination which ray-tracing and pipeline rendering are capable of. Ray-tracing and pipeline rendering can additionally capture the direct diffuse illumination which the radiosity solution includes.

While a radiosity solution does include the direct diffuse illumination of the environment, this is simply the first iteration or “bounce” of the total radiosity solution. If this first iteration is discarded from the final solution, the radiosity renderer produces the global indirect illumination of the environment. If no ambient term is included, this provides a linearly independent lighting calculation which can be simply added to the lighting calculations as defined above. This method is better suited for dynamic environments than other two-pass methods [WCG87][PSV90] in that the global direct component is not incorporated in the first pass solution and the indirect component is not as affected by moving geometry.

The addition of the radiosity contribution readily occurs in image-space as demonstrated by Dorsey et al.[DAG95]. This method, however, can prove more complicated if specular surfaces are present. The radiosity image typically will not contain secondary images as in a mirrored image, and therefore the entire image cannot be added to the pipeline rendering image. This region needs to be first masked to prevent blending of one image containing the specular image (the pipeline rendering one) with one image without specular reflections (the radiosity one). Another solution is to bring the radiosity solution into the pipeline rendering system, and add only secondary specular images there. This produces an image containing the indirect contributions of the scene itself as well as of the specular images of the scene as visible in mirrors, etc. This image can then be directly added to the pipeline rendering image which contains all of the direct illumination effects based on the
linearity of light transport.

In fact, the converse of this image combination method can be used to obtain the indirect radiosity image contribution by differencing the final radiosity image with the image produced after shooting only the emitting patches. This method is demonstrated in Figure 6.13 where the image (a) is generated from a radiosity solution after 2224 patch shootings and the image (b) is generated from shooting only the 24 emitting lightsource patches. Image (c) is the difference of the first two, thereby representing only the indirect contributions. This image can then be combined with image (d) produced with the pipeline rendering methods as described above, producing the last composite image (e).

A more flexible solution is to incorporate the indirect radiosity solution into the hardware rendering process instead of adding it to the final image. The indirect radiosity vertex coloring is rendered using the hardware shading in the first pass of the pipeline process. This replaces the ambient-only rendering stage. This obviates the specular image problem described above since specular images are generated also during the ambient-only stage which the indirect solution replaces. The direct solutions then use the original attribute information for all subsequent calculations, ignoring the radiosity vertex colorings. The hardware blending functions detailed previously perform the composition of the individual illumination effects.

6.4 Total Scene Illumination

The previous sections detailed the individual components of the scene illumination model. It is the combination of these techniques which produce the final images. While some discussion has been made about the individual errors which result from hardware limitations, physical approximations, and performance tradeoffs, it is only through evaluation of the image produced by these methods that we can gauge the effectiveness of these techniques as a system.

Figure 6.14 compares an image produced through the Radiance[War94] system
Figure 6.13: Calculating Indirect Illumination Image
and one produced through the pipeline rendering process. The pipeline image uses
the above techniques with a constant ambient lighting of 0.10 replacing the inclusion
of a radiosity solution indirect component. Image (a) shows the reference Radiance
image which required 220 sec., and image (b) shows the pipeline image generated
in 8.5 sec. Images (c) and (d) are the difference images between the two, with a
gray RGB value of [128,128,128] (from a range of 0-255) representing the zero value
(equal intensities). The RMS error between the two images, as shown in Image (e),
is under 5% over the range of luminance values versus over 10% error for traditional
single-pass rendering.

Further accuracy can be gained using the radiosity-generated indirect component
as described above, however the pre-processing computation cost often outweighs the
increased realism benefit. The indirect component required 2224 patch shootings
for a total of 2343 CPU seconds. While this indirect term is not as effected by
a dynamic environment as the direct components, it can require recomputation if
the scene geometry changes too drastically. This high computation cost, as with
any radiosity-based system, prevents incorporation into a fully-dynamic interactive
system. For a low ambient scene which is dominated by specular reflections, a
constant ambient term may be adequate for interactive perusal.
Figure 6.14: Gaussian vs. Phong Scene Illumination
Chapter 7

Performance

There is a tradeoff between performance and realism in computer graphics. While hardware-based rendering permits real-time interaction, it has been limited in the quality of the image it produces. Ray tracing systems produce very accurate scenes of specular environments, but each image requires significant computation time. Radiosity systems provide accurate 3-D diffuse lighting representations of relatively static environments and additionally require considerable precomputation. Graphically, the relationship between performance, dynamics, and quality is seen in Figure 7.1. Note that this does not include any notion of associated overhead.

Pipeline rendering introduces a series of related multipass techniques which provide added quality. It in essence fills the gap of Figure 7.1 between the hardware-based techniques and the software-intensive processes as well as providing for a more dynamic environment at a high level of quality. Because it relies on multiple rendering passes to add additional levels of detail in each pass, it can easily be tailored to suit the particular user's desired performance-quality needs. This thereby provides a broad spectrum of image quality and realism (Figure 7.2).

We have detailed the coordination between the pipeline rendering techniques including shadow generation and specular surfaces. As both of these features proceed recursively, performance is primarily determined by the number of rendering passes (including both scene and shadow volume rendering); image size has minimal effect.
Figure 7.1: Rendering Relationship
Figure 7.2: Pipeline Rendering Relationship
as compared to ray tracing. For the rendering of Figure 5.1 consisting of 40,463 polygons, the scene required 96 shadow passes and 49 scene passes which together represent 92% of the total rendering time of 16.6 seconds for the 623x942 image. A 1270x995 image required 24.8 seconds. The corresponding Radiance images without textures required 236 and 486 seconds, respectively. All renderings were performed on a single-processor 200MHz R4400 Onyx Reality Engine. A breakdown of the rendering process is seen in Table 7.1. A detailed breakdown of the rendering process for one of the four light sources illuminating the mirrored image is included in Table 7.2. This represents the lighting of the mirror specular image of the scene (view of the bathroom looking through the mirror) from one light source. This is composed of four scene passes, encompassing eight shadow passes.

As can be seen by this data, specular image generation and shadow generation can become very costly for highly specular environments for large depths of recursion. While much of this cost could be eliminated by more accurate viewpoint culling (i.e. Mea82, Air90) for both specular image and shadow generation, some is inherent in the nature of the process. Yet even this cost can be reduced if some sacrifice of image quality or accuracy is permitted. The following discusses this quality/performance tradeoff.
<table>
<thead>
<tr>
<th>Level</th>
<th>Specular Object</th>
<th>Pass</th>
<th>Iterations</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N/A</td>
<td>Shadow Volume Rendering</td>
<td>1394</td>
<td>0.057</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object Rendering</td>
<td>38,839</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>View-in-volume Check*</td>
<td>99</td>
<td>0.007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Misc. Checks</td>
<td>58</td>
<td>0.007</td>
</tr>
<tr>
<td>1</td>
<td>mirror</td>
<td>Shadow Volume Rendering</td>
<td>2778</td>
<td>0.116</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object Rendering</td>
<td>38,839</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>View-in-volume Check</td>
<td>199</td>
<td>0.013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Misc. Checks</td>
<td>117</td>
<td>0.008</td>
</tr>
<tr>
<td>2</td>
<td>shower door</td>
<td>Shadow Volume Rendering</td>
<td>1491</td>
<td>0.053</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object Rendering</td>
<td>38,839</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>View-in-volume Check</td>
<td>188</td>
<td>0.010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Misc. Checks</td>
<td>116</td>
<td>0.008</td>
</tr>
<tr>
<td>1</td>
<td>shower door</td>
<td>Shadow Volume Rendering</td>
<td>119</td>
<td>0.003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object Rendering</td>
<td>38,839</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>View-in-volume Check</td>
<td>57</td>
<td>0.003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Misc. Checks</td>
<td>15</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>1.07</strong></td>
</tr>
</tbody>
</table>

Table 7.2: Single image/single light rendering statistics for bathroom environment.

* Number of objects checked based on which side of specular object.
† Total does not equal the sum of the independent times due to overhead and functions not mentioned, as well as roundoff error.
†† # of Polygons
‡‡ # of Objects
7.1 Diffuse Transport

Section 6.3 discussed incorporating the diffuse lighting calculations from a radiosity shooter to include full global and local specular and diffuse lighting components in the pipeline rendering system. While this produces the most accurate images, this additional step is often unwarranted. While useful in adding specularity or specular transport to a radiosity scene, the diffuse components may provide very little additional information in environments with little diffuse-diffuse transport, low ambient lighting, or dominating specular surfaces. In addition, a highly dynamic environment with significant diffuse-diffuse transport might require several radiosity recalculations which make inclusion extremely costly.

7.2 Shadows

Shadows are known to be an important visual cue for producing realistic and understandable images. In a specular environment or an environment with many light sources, shadow patterns can become extremely complex as seen in Chapter 4. For an environment with $n$ specular surfaces each visible to the others, shadows are highly dependent on the depth $d$ of the recursive light reflections and refractions. As described in Section 4.2, shadow generation is required for all virtual light sources resulting from the specular bounces, as well as shadows which occur from the original light source and are subsequently bent. This requires

$$O(n \left[ \frac{d^2 + 3d}{2} \right] + 1)$$

shadow generations per image per light. Often, however, shadows are desired to simply provide visual cues and some level of realism, and are not required to be totally accurate. In these situations, the number of rendering passes as well as the memory overhead can be greatly reduced. In addition, selective rendering of only significant light sources (and therefore shadow
volumes) can also provide a means to increase performance.

Figure 7.3: Varying Shadow Settings

Figure 7.3(a) shows the lighting passes for a single light source with one reflective and one refractive surface. In Figure 7.3(b), shadow volumes which encounter more than one specular surface are eliminated. This effect is apparent in the shadow pattern on the tub. Figure 7.3(c) shows the original situation without reflected and refracted shadows (objects before the specular surface). Note the light volume is larger because the frame no longer blocks light from reaching the mirror, and the shower door handle no longer casts a shadow through the door. Figure 7.3(d) replaces the refractive shower door with a purely transparent, non-refracting surface.
Table 7.3: Varying Shadow Settings

<table>
<thead>
<tr>
<th>Fig.</th>
<th>d</th>
<th>Spec Shad</th>
<th>2+ Bounces</th>
<th># Passes*</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>2</td>
<td>YES</td>
<td>YES</td>
<td>17/32</td>
<td>21/44</td>
</tr>
<tr>
<td>(b)</td>
<td>2</td>
<td>YES</td>
<td>NO</td>
<td>17/28</td>
<td>21/28</td>
</tr>
<tr>
<td>(c)</td>
<td>2</td>
<td>NO</td>
<td>NO</td>
<td>17/16</td>
<td>21/20</td>
</tr>
<tr>
<td>(d)</td>
<td>2 (1 used)</td>
<td>N/A</td>
<td>N/A</td>
<td>9/12</td>
<td>9/12</td>
</tr>
<tr>
<td>(e)</td>
<td>1</td>
<td>YES</td>
<td>N/A</td>
<td>13/20</td>
<td>13/20</td>
</tr>
</tbody>
</table>

*The first set of numbers represents the actual number of passes made. The second set represents the predicted number based on Eq. 7.1 if no additional visibility testing was performed.

Finally, Figure 7.3(e) demonstrates the original scene with a shadow recursion depth of one instead of two. A summary of the required number of rendering passes is seen in Table 7.3.

7.3 Specular Images

Like shadows, specular images also provide a measure of depth perception as well as added realism. In a complex specular environment, the interaction of specular surfaces can require many rendering iterations. Yet like shadows, it is often not required to have total physical realism. This tradeoff of realism for performance can again be accomplished by reducing the recursive depth. Partial specular surfaces can be selectively rendered with the secondary specular image. Furthermore, simple non-refractive transparency can often replace refractive surfaces as an approximation as we did with shadows in Figure 7.3.

Because shadows are generated within each specular image as well as recursively for each specular surface, the number of specular surfaces as well as recursion depth proves doubly important. For complete shadow rendering, shadow volume passes are required for shadows on each side of every specular surface encountered for the lightsource at that recursive level. For \( n \) specular faces at a recursive image depth
of $d$,

\[ O(n^d + 1) \quad (7.2) \]

images are generated. Again, viewpoint culling can greatly reduce the actual number of passes required.

The images in Figure 7.4 and the timings in Table 7.4 demonstrate the relationship between image quality and performance for varying specular environments. Figure 7.4(a) shows three specular surfaces (mirror, shower door, and floor) at a specular depth of two, and Figure 7.4(b) at a specular depth of one. In Figure 7.4(c), the partial specular floor image has been excluded at a depth of two. This is then repeated at a depth of one in Figure 7.4(d). In Figure 7.4(e), the refractive shower door is replaced with a non-refractive transparent surface at a depth of two.

![Figure 7.4: Varying Specular Settings](image)

(a) 63.5 sec.   (b) 28.0 sec.   (c) 11.4 sec.   (d) 6.9 sec.   (e) 3.4 sec.
Table 7.4: Varying Specular Settings

<table>
<thead>
<tr>
<th>Fig.</th>
<th>n</th>
<th>d</th>
<th>Scene Iter # (sec)</th>
<th>Shad Iter # (sec)*</th>
<th>Time (sec)</th>
<th>Rad Time (sec)</th>
<th>Perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4(a)</td>
<td>3</td>
<td>2</td>
<td>321 (7.6)</td>
<td>760 (10.3, 53.5)</td>
<td>63.5</td>
<td>367</td>
<td>6x</td>
</tr>
<tr>
<td>(b)</td>
<td>3</td>
<td>1</td>
<td>129 (3.1)</td>
<td>304 (18.0, 23.8)</td>
<td>28.0</td>
<td>236</td>
<td>8x</td>
</tr>
<tr>
<td>(c)</td>
<td>2</td>
<td>2</td>
<td>81 (1.9)</td>
<td>160 (7.2, 9.2)</td>
<td>11.4</td>
<td>238</td>
<td>21x</td>
</tr>
<tr>
<td>(d)</td>
<td>2</td>
<td>1</td>
<td>49 (1.1)</td>
<td>96 (4.2, 5.4)</td>
<td>6.9</td>
<td>211</td>
<td>30x</td>
</tr>
<tr>
<td>(e)</td>
<td>1</td>
<td>2</td>
<td>17 (0.5)</td>
<td>24 (2.5, 2.8)</td>
<td>3.4</td>
<td>180</td>
<td>53x</td>
</tr>
</tbody>
</table>

As can be seen, the rendering times range greatly yet many of the images are similar. In an interactive environment, complete rendering may not be needed until motion stops. This method is ideal for a progressive refinement situation, which could substitute from (e) to (d) to (c) to (a) as the necessary frame-rate drops (i.e. the camera slows).

### 7.4 Scene Dynamics

As detailed in Section 4.2.2, shadows maintain object-associativity information. Therefore, unlike in radiosity, moving scene geometry has little effect on rendering times. For the sequence of images in Fig. 7.5 from an animated sequence, the movement of the mirror required the most recomputation (6.1 sec/frame) as it affected virtual light source shadows. The movement of the animated human required less recomputation (4.9 sec/frame). The entire animated sequence of 450 frames required less than 6 hours of rendering time.
Figure 7.5: Varying Scene Geometry and Lighting
Chapter 8

Scene Uncluttering

The methodologies presented thus far have been directed toward increasing scene realism at varying levels of performance. Yet with the added details introduced by additional scene geometry and effects comes an added level of image complexity. It is often desired to concentrate on a particular aspect of a scene with little or no regard to the remaining portions. With current systems able to display vast amounts of data, scene uncluttering becomes an essential tool in a user-friendly environment.

8.1 Background

There has been much use of transparency and clipping throughout computer graphics. In addition to the added realism which transparency can bring, both features are powerful tools for the interactive minimization of environment details. Unwanted or unneeded details can be made clear or eliminated completely through these two methods. These features are readily available in ray-tracing software, yet their functionality is severely limited in hardware-based Z-buffer rendering systems where their use is of potentially more benefit. The following discusses hardware-related implementations.
8.1.1 Clipping Planes

Clipping planes are an essential part of any 3-D graphics rendering pipeline. 3D world coordinates are generally normalized and clipped against a canonical view volume before being submitted to the Z-buffer algorithm during rasterization. In a typical pipeline, clipping is performed in homogeneous coordinates for $Z$ clipping; $X$ and $Y$ clipping may take place after the homogeneous divide for efficiency.

Clipping of the canonical view volume can use extensions of 2D clipping algorithms such as the Cohen-Sutherland and Cyrus-Beck\cite{CB78} methods. Although clipping may be performed on a perspective-projection canonical view volume, it is often convenient to transform it to a parallel-projection canonical view volume and clip against the trivial plane equations:

$$X = -W; X = W; Y = -W; Y = W; Z = -W; Z = 0. \quad (8.1)$$

In some hardware systems, other user-defined clipping planes are provided for clipping in eye-coordinates. SGI provides up to six arbitrary clipping planes for specifying half-spaces based on the user-supplied plane equation. This plane is transformed to eye-coordinates using the inverse of the ModelView matrix, and each vertex is dotted with the transformed plane equation. Negative inner products are clipped. Both user-defined and the parallel-projection clips occur in parallel processors \cite{Cla82} which implement the Cohen-Sutherland algorithm when necessary.

Although these user-defined clipping surfaces do permit some user selection of scene geometry, even a single clip eliminates an entire geometric half-space. Selection of multiple clipping surfaces provides limited control as the resulting clipping region is the interior of up to 12 half-spaces and therefore always convex.

8.1.2 CSG

Constructive Solid Geometry (CSG) and similar boolean operation systems requires geometric clipping for differencing, intersection, and half-spacing. Direct CSG rendering from the object’s binary-tree representation can be accomplished in either
image-order or object-order rasterization. Although a direct image-order rasterizer has been developed [KE84], a more general solution uses object-order rasterization based on Z-buffer algorithms [GHF86]. Kelley et al.’s [KGP+94] extension of [Mam89] stores multiple Z-RGBA values for visibility clipping of CSG objects.

8.2 Clipping Surface

Clipping surfaces are a powerful method for conveying depth information of a scene. As noted in [FvDFH90], dynamic modification of the view volume gives the viewer a good sense of spatial relationships between parts of the environment, as well as serves to eliminate extraneous objects and allow the viewer to concentrate on a particular portion of the world.

As can be seen in the previous section, the DZ-buffer permits two individual Z value comparisons with two distinct buffers. This makes it ideal for implementing a view dependent arbitrary clipping surface. This is accomplished by writing the furthest Z values of the clipping surface to one buffer designated as CLIPBUFFER. All objects are then rendered to the other buffer using normal Z-buffer rendering. The additional comparison is made on CLIPBUFFER so that all rendered pixels have Z values greater than CLIPBUFFER. The entire process is seen in the following pseudo-code:

```c
clipsurface(Segment *clipsurface)
{
    zclear(CLIPBUF, ZMIN); //init to minimum Z
    zfunction(CLIPBUF, ZF_GREATER);
    zfunction(DRAWBUF, ZF_NONE);
    zwrite(CLIPBUF);
    wmpack(0x00000000);  //don’t write color
    draw_seg(clipsurface); //reverse Z sort into CLIPBUF
    zclear(DRAWBUF, ZMAX); //init to maximum Z
    zfunction(CLIPBUF, ZF_GREATER);
    zfunction(DRAWBUF, ZF_LESS);
    zwrite(DRAWBUF);
    wmpack(0xffffffff);    //write color too
    draw_all_segs();       //Z sort, further than CLIPBUF
}
```
For a visual representation of the clipping surface, it is often useful to render the surface in wireframe after all other segments have been drawn and clipped.

### 8.3 Transparency Surface

The arbitrary clipping surface can be a powerful tool for elimination of unwanted geometry. It is often desirable, however, to simply minimize this geometry while still using it for positional reference. Transparency is ideally suited for this purpose; it provides a powerful mechanism for uncluttering complex worlds. In combining the clipping surface detailed above with the correct transparency rendering, an arbitrary transparency surface is achieved. Instead of clipping away geometry on one side of the transparency surface, the enclosing geometry becomes transparent thereby displaying the opaque geometry beneath it. This, like clipping surfaces, permits the user to concentrate on desired aspects of the environment. Correct sorted blending is, however, still required to properly display these relationships.

To perform a transparency clip, an ordinary surface clip is performed as described above. The back-to-front transparency process is then performed, using the clip rendering in place of the initial opaque rendering. The resulting DRAW buffer from the clip is used as the initial PREVIOUS buffer in the transparency routine, and the subsequent transparent layers are rendered on top of the clip rendering. Rendering the clipping surface in wireframe after the clip will produce a valid image as the transparent layers will layer on top of it also.

### 8.4 Software Emulation

All of the clip and transparency surface features were implemented in software on the SGI GL platform. Each surface was individually rendered to its own bounding-box size buffer and Z-comparison and composition was performed on a pixel-by-pixel basis. Even with software Dual Z-buffering of the bounding box of each drawn face,
Figure 8.1: Clipping/Transparency Surface

(a) Clip Surface  (b) Cut-away area
(c) Clipped Area  (d) Transparent Area
the rendering proved quite efficient due to the simple buffer swap provision. The clip in Fig. 8.1 of over 14000 polygons was performed in under one minute real-time on an SGI VGX 33MHZ R3000 system for an NTSC size window and under one and a half minutes at full-screen. The transparency clip took under 6 minutes. Front-to-back had slightly lower performance due to the additional software implementation of the unsupported accumulation buffer function.

A hardware implementation would have eliminated the cost of these pixel operations and all cost would have been in the multiple iterations, which is governed by the number of overlapping transparency layers. A typical unsorted Z-buffered frame of the same scene is rendered in 150ms. Using this as the metric, the transparency clip which contains a transparency depth of 12 would render in approximately 1.8 seconds assuming both Z comparisons could be made in parallel. Only minute details were added in the latter transparency stages; therefore, for many applications the full depth may not be necessary and the required number of iterations reduced.
Chapter 9

Extensions

We have presented a collection of hardware-based multi-pass rendering techniques which can provide many ray-trace quality effects in a user-interactive dynamic environment. While these techniques are not applicable to all types of environments and are not desirable when absolute physical accuracy is necessary, they do provide a wide range of solutions in terms of cost versus performance. In order to demonstrate the value and application of the pipeline rendering system, a discussion of the current limitations and possible extensions follows.

9.1 Limitations

While the techniques described can produce fast, complex images, they suffer from several shortcomings based on their use of empirical methods, hardware limitations, and approximations. Because pipeline rendering relies on multiple viewpoints refractive and reflective surfaces must be planar. Shadows may suffer from aliasing effects due to the use of image space precision in the calculation. In addition, hardware lighting (typically the Phong model [Pho75]) is used for the illumination model, which is widely known for its inadequacies [War92]. In the same regard, the system is hardware dependent on the number of stencil and accumulation bits, as well as the viewport-screen transforms supported.
The rendering phase is also very time-dependent on the complexity of the environment as well as the recursion level for refractions (reflections). For an entire scene rendering at a given shadow and specular recursive depth,

\[ O((n^{d+1} + n) \left[ \frac{d^2 + 3d}{2} \right] + n^d + 1) \]  

shadow passes are required for complete scene rendering and illumination.

For complex recursively refractive and reflective surfaces at an arbitrary depth, this expense can quickly become prohibitive even when compared to ray-tracing. This method is more applicable for environments with few mirrored surfaces such as in a virtual building or where recursion is limited to a minimal depth.

For scenes with a limited number of planar refractive and reflective surfaces, or with a low recursive depth, this system is very effective even with minimal hardware support. The system currently runs effectively on an Iris Indigo XS24\textsuperscript{TM} with 8 stencil bits and a 48 bit accumulation buffer.

## 9.2 Hardware Extensions

Shadows, reflection, and refraction are all provided using the currently available rendering pipeline support. These methods require stencil bit planes, an accumulation buffer, alpha channels, and a standard Z-buffer implementation. For refractive transparency, additional pipeline transformation control (i.e. 2-D image mapping) would greater facilitate implementation. The projective texture mapping, while partially hardware-supported, could also benefit from additional matrix stack operations as with the refractive surfaces. Even without changing the Phong lighting model, greater realism could also be accomplished through better hardware support for it through a more dynamic range of glossiness exponents.

The \(\text{DZ-buffer}\) (and \(\text{TZ-buffer}\)) extension is hardware implementable with little or no performance loss. This facility requires only one (two) additional \(Z\)-value comparator which could be implemented in parallel with the current comparison
and additional Z-buffer bit-planes. With this evolutionary extension, sorted transparency compositing is possible at interactive rates on a pixel-by-pixel basis without the pixel-by-pixel buffer operations of other multiple Z-buffer methods. This compositing can be performed both back-to-front or front-to-back with current and slightly modified stenciling functions and a slight modification of the accumulation buffer[HA90] return function. The DZ-buffer extension also provides arbitrary clipping surfaces which combined with the transparency methods enables arbitrary transparency probes of the environment geometry.

This work is not only an introduction of new methods, but it also serves as a platform from which to incorporate other hardware techniques to build a complete interactive renderer, as described in [tHKT86]. There are many issues which could be addressed at the hardware level, such as more control for direct manipulation of the rendering pipeline as well as more complex hardware lighting models. With advanced hardware features finding exotic uses in producing effects such as texture-mapped shadows, this stresses the need for greater flexibility in user access to these features.

Additional hardware support would provide greater facilities for creating more complex images. Additional pipeline control such as viewport transforms or additional fog features would enable distorted refractions in conjunction with translucency. Multiple accumulation buffers would enable handling partially reflective and partially refractive surfaces, instead of merely switching at the critical angle. While many of these hardware support features are not found on our chosen hardware platform, some are readily available in other graphics architectures.

9.3 Hardware Platforms

The rendering techniques we have introduced are based on implementation of a serial pipeline architecture such as the SGI platform[Ake89][Ake93a]. While the SGI rendering pipeline is currently dominant in 3-D graphics hardware rendering,
there are alternative systems which have addressed 3-D rendering by other means. Although many different graphics rendering architectures have been developed, most of the rasterization approaches can be categorized in regards to their concurrency in both the front-end (model traversal) and back-end (primitive rasterization). To date, most systems rely on serial traversal with parallelism found only in the rasterization stages. This back-end parallelism can be found in both object-order rasterization (i.e. Z-buffer, depth-sort) and image-order rasterization (i.e. scanline algorithms). In addition, hybrid systems exist which make use of both object-order and image-order parallelism.

We examine these architectures with respect to their rendering capabilities and limitations. We then address using our pipeline rendering methods to expand the rendering capabilities on the various systems, and suggest ways to overcome inherent architectural limitations preventing even greater realism.

9.3.1 Object-Order Rasterization Systems

Object-order rasterization systems perform rendering object-by-object, without regard to which pixels they affect. Parallelism is achieved by partitioning image memory so that several processing elements simultaneously rasterize the same object primitive.

This SGI platform[Ake93a] provides serial pipelined traversal of the display model; parallelism is provided in its object-order image-parallel rasterization and multi-stage pipeline. Parallel span processors each handle a fraction of the screen columns. Pixel-Planes[FP81] also provides image-parallel rasterization, although it replaces SGI’s parallel span rasterization with massive pixel-level parallelism in the rasterization of serially-processed primitives through the use of SIMD pixel processors. Pixel-Planes 4 obviates the polygon rasterization limitations of the SGI platform by providing scan conversion which is independent of a polygon’s size. Systems also exist which have made enhancements and modifications to the above Gouraud-shaded
polygon-rendering architectures. Real-time antialiasing, alpha-blending, and texturing have become commonplace through frame-buffer extensions, and some systems (i.e. Stellar GS1000[ABM88]) has generalized this concept by providing virtual pixel maps on which a variety of post-processing can occur.

While these systems do generally provide fast antialiased polygonal rendering, they have little direct support for the rendering features detailed in our methodology. Shadows are supported on the SGI platform through both texture mapping and shadow volume stenciling, however neither is readily available for true interactive systems. The shadow buffer method requires fast recomputation of shadow textures, which is currently not supported. Shadow volumes currently require duplication of effort in that front-face/back-face determination is required for stencil-function specification, yet this is automatically performed by the hardware rasterization algorithm. Great speedup could be achieved with a stencil function based on primitive normal direction.

The Stellar GS[ABM88] family provide a much more open architecture for implementation of our multi-pass techniques. In fact, the DZ-buffer features we describe are based on a multi-Z-buffer implementation on the GS1000[Mam89]. Their Virtual Pixel Map concept permits definition of a pixel to be whatever the application requires. This, in essence, permits frame buffers, Z-buffers, stencil buffers, accumulation buffers, and any other type of related pixel information to be stored, operated on, and retrieved. The flexibility of this system provides a superior testbed for a multi-pass implementation of image-masking and composition techniques. The openness of this platform does, however, have its cost; display requires transfer of image data from the virtual pixel maps to the frame buffer, requiring extra bandwidth and time.

Despite varying support for our advanced rendering features, image quality on each system is limited by more basic factors. Although the object-order rasterization systems have vastly different architectures, there are many similarities in regards to
the physical accuracy of the images they can generate. Most of the mentioned systems rely on a Z-buffer to perform visible surface rendering, and therefore may suffer from coincident polygons resulting from roundoff error. Most also use a hardware-based Gouraud-shaded Phong illumination model and perform their calculations in RGB space, thereby severely limiting their luminance dynamic range.

Image resolution can be improved, however, by addressing the limitations of each of these factors. Z-buffer resolution is dependent on the depth of the bitplanes, where each additional bit doubles the resolution. Current architectures such as Silicon Graphics' RealityEngine² provide 32-bit Z-buffer depth compared to 24-bits in their older systems [Ake93a], thereby producing 256 times the resolution. Hierarchical Z-buffers[GK93] have also been proposed to minimize the floating-point limitations. In addition, aliasing problems with hidden surface elimination introduced by subpixel masks have been addressed with expanded Z-buffer systems[SS93][Car84].

Other improvements can be made through use of better illumination and shading models. As demonstrated in Section 6.2, even the current Phong model can be made more physically based. Other lighting models have been developed for improved speed [Sch94] and accuracy[HTSG91]. Much better shading methods exist including Phong shading[Pho75][BW86] which, despite interpolation inaccuracies such as from perspective foreshortening, can produce much better specular highlights especially for large primitives. This itself can reduce the need for polygonal tessellation and thereby recapture some performance lost by the costlier method.

Finally, the calculation space of these illumination evaluations can be expanded to better handle the wide disparity of real-world luminance values. Typical radiance values in a scene can often vary over a range of 1000:1[War91] with real world luminances ranging from $10^{-6}$ to $10^4 \, cd/m^2$[TR93]; however most lighting calculations are performed in 8-bit integer space per color channel with monitor luminances in the range of 1 to 100 $cd/m^2$. Ward [War91] presents a simple extension for 32-bit real valued pixels (already available in most architectures) which have a far greater dynamic range.
9.3.2 Image-Order Rasterization Systems

Whereas object-order rasterization performs rendering based on object-traversal, image-order rasterization systems perform rendering on a pixel-by-pixel basis. Little or no regard is made for neighboring pixels.

Variations on scanline rasterizers have been implemented in hardware which make use of screen partitioning [Fuc77] [NIM+84] as well as scanline partitioning [KWG92]. Systems such as the NASA II flight simulator [BE89] have focused on object-parallel rasterization where each primitive is assigned to object processor, and a priority-multiplexor combines the resulting rasterizations. Other primitive-processor strategies have been designed, including pipelined architectures (i.e. by Cohen and Deme trescu [Dem80]) which successively filter color and Z streams, and tree-structured architectures (i.e. by Fussell [FR82]) which hierarchically merge these streams. Other image-order systems have abandoned primitive-based rasterization completely for ray tracing-based architectures (i.e. Ray Casting Machine [KE84], LINKS-1 [NOK+83]), which access primitives only on a per-need basis.

Because most object-parallel image-order systems perform image composition based on Z values and color blending, their structure is very similar to the multi-pass techniques presented. There is, therefore, a natural extension of our methods for object-parallel architectures. Special attention, however, must be paid to composition order, especially for recursive specular images. Our methods rely on depth-first traversal of the images, which does not directly result from object-parallel rendering. Multiple-light source illumination and accumulation, as well as shadow and view jittering, does lend itself to this parallel implementation, as these processes already are focused on image composition.

Ray tracing and partitioning image-order architectures which do not support object parallelism are not as well-suited for our multi-pass techniques in that they make no use of primitive association. Ray tracing systems do, however, typically have full support of the features we detailed in that they proceed in an object-space
tracing, and therefore our methods provide little or no benefit.

As these systems operate on an independent per-pixel basis, they all tend to suffer from problems inherent in such discrete rendering algorithms. This includes aliasing artifacts such as “jaggies”, jagged highlights, and Moiré patterns in textures [WW92]. These aliasing problems, as in distribution ray tracing, can be addressed by supersampling, adaptive sampling, and stochastic sampling. In addition, the problems (and solutions) addressed in Section 9.3.1 for object-order systems also apply for many systems.

9.3.3 Hybrid Systems

Hybrid systems attempt to make use of both object- and image-parallelism at various stages in the architecture. This usually entails object-parallel rasterization in separate screen regions or yet-to-be-merged frame buffers.

Hybrid graphics systems generally either exploit bucket sorting (i.e. Pixel-Planes 5 [FPE+89]) to sort primitives according to screen region for parallel processing, or use image composition (i.e. PixelFlow [MEP92]) to combine separately generated frame buffers which contain the distributed primitive renderings.

The Pixel-Planes 5 system sorts primitives into “bins” corresponding to patch-sized regions of the screen. The transformation engine maintains the full list of sorted primitives, which are transformed, sorted and stored, and assigns all available Renderers the corresponding bins for their screen patch. Once a renderer is completed with a patch, it transfers the color information to a “backing store” and is assigned a new patch and associated bin. While this bucket sorting approach is compatible with our multi-pass method, the backing store approach is not. The Pixel-Planes 5 backing store typically maintains only color information; Z values and other pixel-associated information is discarded by the Renderer. This precludes the masking and color overlays which comprise our methods. Fuchs does mention other implementations of the backing store for volume rendering, so an extension for
additional bit-planes should be feasible.

In contrast, the PixelFlow system also performs object-parallel rendering, but does not rely on screen partitioning. It instead assigns a separate processor for each primitive, and renders that primitive to a separate frame buffer. As with the processor-per-primitive approaches described in Section 9.3.2, the PixelFlow image composition method is directly analogous to our multi-pass image method. This system, however, performs _deferred shading_, which composites pixels based on attributes instead of color. This provides an even more general framework for our methods as composition operations need not take place in RGB space.

With specialized rendering processors, these systems have been able to implement some more advanced rendering features, such as Bishop and Weimer’s Fast Phong Shading [BW86]. In addition, the generality of the processor-enhanced memories permit pre-sorting transparent polygons when possible, and using a multi-pass approach approach for complex intersecting surfaces. They do, however, introduce some additional problems and limitations in addition to those described in Sections 9.3.1 and 9.3.2. This includes shadow volume processing, which does not readily fit into Pixel-Planer screen-division bucket sorting strategy [FPE+89].
Chapter 10

Conclusion

With multi-pass pipeline rendering, we have presented a platform for bridging the gap between static off-line rendering systems and dynamic hardware-based graphics. We demonstrated a practical implementation of shadow and light volumes and incorporated this into a recursive paradigm permitting interaction with specular surfaces. This includes the specular direct component similar to Radiance’s[War94] virtual lights for planar specular surfaces. We showed an extension to projected textures to approximate complex material transmission, and have tried to wrestle as much physical realism out of the lighting model itself without compromising performance. Consistent with the broad spectrum of achievable quality, we also presented a method to even include indirect lighting effects. Even with minimal viewpoint culling, this pipeline rendering method demonstrated typical performance rates 5 to 50 times that of ray tracing for our test environments. Future work will focus on taking full advantage of available culling strategies for dynamic environments.

10.1 Contributions

We have described a series of techniques for adding realism to interactive environments and making these environments more visually comprehensible. These techniques have the common thread of using the hardware rendering pipeline itself to
produce illumination effects commonly found only in non-interactive renderers such
as ray-tracers. These techniques exploit implicit pixel-parallel operations, and there-
fore have application not only in the current multi-pass pipelined method, but also in
a multi-threaded operation on rendering architecture which supports such coherence.

In summary, the primary individual contributions of this work:

- Specular surface rendering, including:
  - Recursive specular surfaces through secondary viewpoint image mapping.
  - Corrective image transform for refractive surfaces.
  - Back-to-front and front-to-back transparency blending through an ex-
tended (Dual, Tri) Z-buffer.
  - Light-scattering (i.e. translucent) specular surfaces.

- Practical shadow volume rendering techniques.

- Recursive specular shadow and light volume rendering via virtual light sources.

- Light volume filtering through texture projection.

- Coordination of shadow volume and specular surface rendering techniques.

- Presentation of pipeline-based illumination model, including:
  - Global Direct Illumination: jitter and accumulation effects.
  - Local Direct Illumination: Gaussian-fit Phong model.
  - Indirect Illumination: integration of indirect (radiosity) component.

- Scene uncluttering through geometry clipping and transparency surfaces.
10.2 Future Work

While these contributions have been demonstrated in a practical, realistic, and dynamic rendering system, there are future extensions and areas of research to be explored. While many of these extensions rely on hardware-based innovations, some derive from research from other rendering techniques. Two of the most important topics of research for interactive rendering include:

- Fast intersection methods for view-in-volume checks and for subdivision for partial intersection of silhouette faces with specular surfaces.

- Viewpoint culling methods for dynamic environments.

Intersection determination is a much-studied area of research for ray tracing systems[Glä89] as well as for collision detection[GF90]. Viewpoint culling and environment partitioning methods are used in architectural walkthrough systems[TS91][ARB90] and in ray tracing subdivision[FTI86] strategies, but most of this work relies on having a static environment or limitations on the geometric layout. Use of these methods to aid in visibility checks between specular surfaces can also be used to reduce the number of scene renderings.

Other extensions focus on the rendering quality of the methods and include anisotropic reflections[War92] which could be simulated based on the reflecting plane’s orientation. In this vein, the lighting model itself could be further modified to be modulated by the surface orientation to include grazing-effect specularity.

In addition to the improvements which are possible in speed and rendering, further research is needed into the perceptual aspects of the constituent effects in a progressive refinement implementation. More specifically, the importance of each of the described effects at varying levels of detail and recursive depths needs to be investigated based on psycho-visual perception metrics. While we have gained some insight into the desired tradeoffs (such as the number of specular surfaces being
generally more important than the recursive depth), we have not focused on ascertaining the qualitative weight of these varying factors based on given timing criteria. As our framework mirrors a progressive refinement approach, adjustment of the system parameters is critical in creation of the most realistically-perceived image in a interactive application. Adopting this system to real-time constraints is still an open-ended topic.

This work also demonstrates the need to develop more open graphics architectures which permit pipeline and image control while providing parallelism to exploit the independent pass rendering and image coherence of this technique. As described in Section 9.3.1, this technique is readily applicable to both image and object parallel architectures, and especially useful in virtual pixel architectures such as the extinct Stellar GS1000 family. A more concentrated effort on an open transformation pipeline, a better illumination model, and increased pixel operation support would potentially provide for greater realism than the focus on simply more polygons and faster texture mapping. Exploiting the inherent parallelism of our methods further makes this approach desirable for increased realism in future hardware systems.

10.3 Conclusion

In summary, we have focused on using multi-pass pipeline-based rendering techniques from existing or slightly modified architectures to provide highly interactive, comprehensible, and realistic environments. With the high investment in existing architectures both monetarily and in terms of the software base, the motivation exists to use the provided platform for greater realism and interaction.

While there will always be a need for complex, very accurate rendering packages, many situations require fast, approximate, solutions. The techniques outlined in this thesis provide such solutions for creating realistic illumination features in complex, interactive, and user-comprehensible environments.

This work intends not only to demonstrate the quality of effects achievable
through pipeline rendering, but also to serve as a call for more focus on the use of the graphics hardware to perform realistic rendering. Indeed, many more extensions of this methodology are possible with current hardware, from achieving a better lighting model through individual vertex normal modulation to use of multiple processors. With other hardware platforms and future hardware systems, parallel rendering pipelines may be able to exploit the independent nature of our multi-pass process. In this vein, we also call for more open, accessible hardware pipelines which provide access in ways which the developers may never had imagined or intended.
Appendix A

Projective Maps

2-D Quadrilateral Projective Map

The mapping of a quadrilateral to quadrilateral can be accomplished by composing the mapping of a quadrilateral-to-square with a square-to-quadrilateral. The two mappings are adjoints of each other symbolically, and therefore only the square-to-quadrilateral mapping\(^1\) will be given:

\[
M_{sq} = \begin{bmatrix}
    a & d & g \\
    b & e & h \\
    c & f & i
\end{bmatrix}, \quad \text{(A.1)}
\]

where

\[
g = \begin{vmatrix}
    \sum x & \delta x_2 \\
    \sum y & \delta y_2
\end{vmatrix} / \begin{vmatrix}
    \delta x_1 & \delta x_2 \\
    \delta y_1 & \delta y_2
\end{vmatrix} \quad \text{(A.2)}
\]

\[
h = \begin{vmatrix}
    \delta x_1 & \sum x \\
    \delta y_1 & \sum y
\end{vmatrix} / \begin{vmatrix}
    \delta x_1 & \delta x_2 \\
    \delta y_1 & \delta y_2
\end{vmatrix} \quad \text{(A.3)}
\]

\(^1\)This mapping is also equal to the perspective transform of the camera rotation with respect to the normal of the refracting plane, which is available directly from the matrix stack.
\[
\begin{align*}
   a &= x_1 - x_0 + g x_1 & d &= y_1 - y_0 + g y_1 \\
   b &= x_3 - x_0 + h x_3 & e &= y_3 - y_0 + h y_3 \\
   c &= x_0 & f &= y_0
\end{align*}
\]

(A.4)

3-D Quadrilateral Projective Map

Given the 2-D quadrilateral projective transform

\[
\mathbf{M}_{22} = \begin{bmatrix}
   a & d & g \\
   b & e & h \\
   c & f & i
\end{bmatrix},
\]

(A.5)

we create the 3-D transform

\[
\mathbf{M}_{32} = \begin{bmatrix}
   a & d & 0 & g \\
   b & e & 0 & h \\
   0 & 0 & 0 & 0 \\
   c & f & 0 & i
\end{bmatrix},
\]

(A.6)

which clears depth values and disables Z-buffering and fog.
Appendix B

Refraction Approximation

Linear Refraction Approximation

Heckbert and Hanrahan [HH84] base their approximation of the refraction transformation on limiting its scope to paraxial rays. For these rays, objects through a refractive index $\eta$ appear to be $\eta$ times their actual distance. Included is a reproduction of their diagram demonstrating their approximation as a scaling transformation perpendicular to the plane:

$$ P = P_t + (\eta - 1)(LP_t)N = M_tP_t $$

where $L$ is the coefficients of the plane equation, $N$ is the normal to the plane, and $P$ and $P_t$ are the real and refracted points.

The corresponding transform $M_t$ gives the paraxial approximation for the virtual focus point. This transform represents the scaling transformation perpendicular to the plane.

$$ M_t = \begin{bmatrix} 1 + \lambda A^2 & \lambda AB & \lambda AC & \lambda AD \\ \lambda AB & 1 + \lambda B^2 & \lambda BC & \lambda BD \\ \lambda AC & \lambda BC & 1 + \lambda C^2 & \lambda CD \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (B.1) $$

where: $\lambda = \eta - 1$

Note that because this transformation is indeed perpendicular to the plane, the
reflection transform $M_r$ is simply determined by substituting $\eta = -1$.

\[
M_r = \begin{bmatrix}
1 - 2A^2 & -2AB & -2AC & -2AD \\
-2AB & 1 - 2B^2 & -2BC & -2BD \\
-2AC & -2BC & 1 - 2C^2 & -2CD \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad (B.2)
\]
Bibliography


