Practical Interactive Lighting Design for RenderMan Scenes

a thesis
submitted to the department of Computer Science
of Stanford University
in partial fulfillment of the requirements
for departmental honors in Computer Science

Jonathan Millard Ragan-Kelley
May 2004

Abstract

This thesis develops a novel technique to enable interactive rendering during lighting design of feature film-quality computer graphics specified in the industry standard RenderMan Interface. This technique is designed to maximize the fidelity of the preview to the results of final-frame rendering, while achieving interactive performance.

The technique presented achieves interactive performance through three critical optimizations. First, it dramatically reduces the computational cost of rendering sequential lighting previews by caching all light-independent computation and only recomputing the portion of the shading which depends on the lighting parameters being edited by the user. Second, it improves performance in common film scenes containing many lights by caching the shading contribution of all inactive lights and only recomputing the shading for the light currently being edited. Third, it achieves extremely high performance on re-rendering computation by mapping it to efficient data-parallel computations which are executed on programmable graphics hardware.

This technique is supported in practice by a novel RenderMan Shading Language compiler which presents an accurate and automated mechanism for the creation of such interactive lighting design previews from arbitrary RenderMan scenes. The compiler employs global data flow analysis to compute a precise lighting specialization of arbitrary RenderMan shaders. It uses this analysis to generate complementary precomputation and re-rendering shaders. The precomputation shaders are executed in place of the original shaders in a rendering of the scene through the final software renderer. The re-rendering shaders are executed in real-time in a generalized interactive lighting design framework which employs the techniques presented herein.

This technique accelerates preview rendering for lighting design by more than three orders of magnitude on real-world scenes, while requiring no modification to existing production pipelines and maintaining complete fidelity to the final results being previewed.

Acknowledgements

This thesis represents the culmination of the first stage in a journey of intellectual exploration and discovery which began in earnest over five years ago. When I was a junior in high school, Matt Pharr and Pat Hanrahan were at once both generous and intrigued enough to talk with me at length about Stanford, the graphics lab, and the seeds of my first research ideas. They provided encouragement and insight which helped bring me to Stanford and led me to discover my greatest intellectual passion in computer science.

From the moment of my arrival at Stanford, they have each taken great risks on me and my research. Pat generously included me in his research group from my first year — before I could even program — and encouraged me to begin my own research project, which would eventually lead to this thesis, just nine months after I arrived. Pat's unmatched knowledge, insight, and experience have taught me more than I could ever have hoped, and the incredible breadth and intensity of his intellectual curiosity have perpetually inspired me to push my exploration further.

Matt has long encouraged my work, and he has served as a mentor and friend through much of my four years. He took a risk on me and this work to invite me to his start-up, Exluna, in 2002. While there, Matt, Doug Epps, and Craig Kolb made me an integral member of their tight-knit team from the moment of my first arrival. I greatly appreciated the opportunity to work with and learn from them, as well as Nick Triantos, Larry Gritz, and others.

At Stanford I have been fortunate to be surrounded by many of the greatest minds, present and future, in computer graphics research. Bill Mark and Kurt Akeley shared astonishing insight with me and others during their time at Stanford. My senior colleagues — in particular Ian Buck, Tim Purcell, and Pradeep Sen — have inspired me and encouraged my work from my first arrival at Stanford almost four years ago. More recently, I have greatly appreciated the intellectual rigor and inspiration of Kayvon Fatahalian and Mike

Houston — they are the colleagues with whom I most regret not having the opportunity to work more closely as I head east to pursue graduate studies.

I was exceedingly fortunate to work alongside office-mate John Owens in my first summer at Stanford. Since then, he has maintained a flattering and inspiring interest in me and my work. Like Matt, he has been a great mentor and friend. His advice and encouragement have contributed in no small part to my successful career at Stanford.

Doug Epps at Tippett Studio and Dan Goldman at ILM have provided tremendous support for my work, championing it internally in their studios and offering me most helpful insight. Doug provided the test scene used in this paper and throughout much of my development, and they both worked hard to provide excellent shader profiling data under extreme time pressure and in spite of frustrating legal hurdles. My results would be nothing without their help. Christophe Hery and others at ILM also offered insight and questions which challenged my thinking and helped shape this project into what it is today.

The compiler implemented in this work is built atop a parser from the open source Aqsis RenderMan implementation and employs the Banshee analysis engine to efficiently implement its global dataflow analyses. Alex Aiken taught me most of what I know about compilers, and he and John Kodumal offered crucial advice in my effort to apply dataflow analysis to RenderMan shaders. Thanks to them, my ideas are now a working reality.

Thanks to all my wonderful friends at Stanford. Thanks to Jeff Mancuso for being a fantastic friend who kept me sane during the most trying and stressful times in the process of this research. Thanks to Dave Chan for being a great friend, almost naive in his most selfless generosity. Thanks to Eric Silverberg for teaching me to love this wonderful school. To everyone, thanks for making these at once the most challenging, exciting, humor-filled, and entertaining four years of my life.

And, above all, thanks to my family — to my parents, Raggs Ragan and Robert Kelley, my grandfather, Randall Ragan, and my uncle, Pete Ragan, and to everyone else — for their intense lifelong interest, encouragement, and support of learning and academia. Without their support, both personal and financial, I would never have had the life-changing opportunity to come to Stanford, and without their example and encouragement I would have lacked both the faculties and the interest to achieve what I have. And finally, thanks to my brother, Min, for always sharing in the excitement of learning about and building great new things — I am proud of the wonderful person you have grown up to be, and know your great heart and incisive mind will take you even further than I.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern special effects and animation production requires extensive digital content-creation work. Content-creation is extremely human-labor intensive, and the expense of this labor limits the quality and dominates the cost of computer graphics production. Production pipelines are traditionally divided into separate modeling, texturing, animation, and lighting and shading stages. Over years of advances in real-time graphics hardware, modeling, texturing, and animation tasks have all become highly interactive on commodity graphics processors. However, the complexity of cinematic lighting and shading computations has typically required technical directors (TDs) to render final film frames — at a cost of tens of minutes to multiple hours — to preview even a single parameter change as they work. Thus, lighting design is perhaps the most critical bottleneck in computer graphics production.

While software rendering systems can be optimized for such work, they are still far from interactive when bearing the full complexity of film scenes — scenes complex enough to routinely require many hours to render a single frame on state-of-the-art computers. Furthermore, software rendering systems inherently fail to take advantage of the rapid development of hardware graphics systems, since these chips still are not flexible enough to render complex film scenes. Even if a reasonably high-quality rendering can be created with a modern graphics processor, the look design process is inherently dependent upon the intricacies of the specific graphics pipeline used in the final renderings. Therefore, any practical interactive look design tool must replicate the exact results of the software rendering pipeline used for final rendering.

Prior work has been done to accelerate this process by precomputing all non-light-dependent aspects of the scene and storing the results in an intermediate deep-framebuffer,

as well as caching the contribution of all lights other than the one currently being modified. However, this work was all on software systems, and hence failed to take advantage of the extremely high performance of programmable graphics hardware for shading calculations. Further, it was limited to a small selection of pre-made shaders which were hand-specialized to separate lighting-dependent from lighting-independent computations. Artists and studios have refused to adopt all prior attempts because they have never yielded sufficient performance to make lighting design genuinely interactive while still providing a usefully accurate preview of their complex RenderMan content.

We implement a three-part system which overcomes the limitations that have crippled adoption of earlier technologies in real-world production. We automatically specialize RenderMan shaders to precompute a deep-framebuffer of non-lighting dependent computations. We then cross-compile the lighting-dependent portion of these RenderMan shaders to interactively compute light-dependent shading from real-time light positions and the precomputed deep-framebuffer on programmable graphics hardware. Finally, we execute these cross-compiled shaders in a generalized interactive lighting design framework using modern GPUs.

Our technique achieves smooth interactive rendering performance when previewing feature film-level RenderMan scenes. It does so while achieving results which are mathematically identical to the final frames output by the software rendering pipeline. Finally, it creates previews automatically, from arbitrary RenderMan scenes and shaders, allowing it to be adopted directly into existing real-world production pipelines.

Outline

This thesis is presented in six subsequent parts. In the first of these we present technical background in the application areas addressed and the theory applied in this research, including related prior work. Next, we describe our approach in greater detail. Third, we present the full details of our implementation. We then present and evaluate the results of this implementation. Fourth, we further analyze solutions to several specific challenges which arise in implementation. And finally we explore and analyze the major challenges and limitations of our approach and its implementation. We subsequently conclude with the status of our current work and present directions of future research.

Chapter 2

Background

In order to understand properly the challenges, design decisions, and unique contributions of our effort to solve the lighting design problem, we must first explore in some detail the fundamental process of digital image creation and the real-world practice of content creation using this technology. We will then introduce prior attempts to solve the lighting design problem, as well as prior work on computational specialization, particularly as it relates to graphics.

2.1 The Rendering Process

Rendering is the process of synthesizing digital images of a virtual scene as it would be observed by a virtual viewer. These images are represented as 2-dimensional arrays of color pixels, each consisting of 3-component vectors of red, green, and blue spectral intensity values, and the scene as a 3-dimensional space. The scene consists of a viewer, a set of lights, and set of geometric primitives with locations and orientations in space. These primitives may be individual polygons or meshes of polygons, or they may be higher-order surfaces with more complicated mathematical representations.

In order to understand our manipulation of the rendering process, it is instructive to consider rendering as a data flow process consisting of several stages of computation, each of which generates the data to be process by the subsequent stage. Each of these stages has different computational characteristics and operates on different data types.

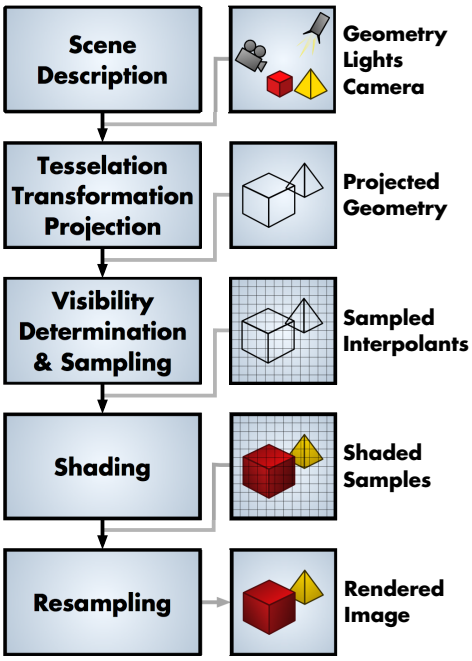

Figure 2.1: The rendering process

Geometry Setup & Transformation The first step in synthesizing a 2-dimensional image of a 3-dimensional scene is the transformation and projection of the scene geometry into the viewpoint for which the image is to be generated. Objects are transformed into the relative coordinate space of the viewer and projected onto the plane of the image — analogous to the projection of light through a lens onto the cornea of the eye or the film plane of a camera — using geometric transformations and perspective projection. Higher-order primitives must generally be tessellated into polygonal meshes for processing.

Geometry setup and transformation operate on the large, often irregular structures in which geometric objects are described. Tessellation of high-order surfaces may require regional access over a moderately large amount of data, and generates an immense expansion at its output, producing a large, irregular data structure.

Visibility Determination & Sampling After the scene geometry has been processed and transformed into image space, the renderer must next determine what point in the scene corresponds to each sample location in the image. Z-buffer or ray tracing algorithms determine what geometric primitive is nearest to the viewer and therefore visible at each

pixel in the image, and the parameters of the corresponding surface are interpolated across the primitive to generate a point sampled set of interpolants at each sample location.

The visibility and sampling stage operates on a large, global, irregular input data stream, consisting of transformed and tessellated geometry, and maps it to a fixed, regular output stream of interpolants.

Shading & Lighting Once the visible point is known for each sample, the renderer computes the perceived color at that sample, corresponding to the light reflected from the scene onto that point on the image plane. This color is computed by the shading model of the given surface, a function which determines the perceived color from the interpolants at the corresponding point, the properties of lights casting onto that point, and the direction from the viewer along which that point is viewed.

The shading computation is a local reduction. It performs a one-to-one mapping from sampled interpolants to colors, accessing only the interpolants of the sample it is shading, and generating a single color from the many interpolants at that sample. Our technique exploits this regularized data flow by only interceding in this stage of the rendering process. This allows a highly regular cache structure and directly enables our efficient implementation on parallel graphics hardware.

Resampling Finally, the shaded, lit samples are resampled into the final pixels of the image. The color output into each pixel is determined by filtering over one or more shaded samples. This generally involves spatial or temporal antialiasing of the finite-resolution output image, but, in the simplest case, each pixel of the final image corresponds directly to a single shading sample.

The end result of the rendering process is a digital image of the virtual scene as it would be perceived by the viewer.

2.2 Computer Graphics Production

Before a computer can synthesize digital images through rendering, a team of artists must full model and describe the virtual world. In practice, computer graphics for feature film effects and animation are produced in a deep, assembly-line-like production pipeline consisting of multiple artists in highly specialized roles, at each stage designing specific aspects of

the content of a scene or shot. Though the content produced for special effects and animation differ in style, large studios in both genres tend to follow a similar pipeline, consisting of four main stages of interest, mirroring the four aspects in which the scene is designed.

Shape: Modeling

The shape of each object in a scene is modeled geometrically in a process akin to sculpting a physical form. The models in film quality scenes generally contain high geometric complexity, but each model can be constructed independently and rendered very quickly at low quality using common OpenGL hardware. The layout of multiple objects in a shot can be performed primarily with low-resolution proxy geometry, allowing fast feedback.

Color: Texturing

Textures make up the non-geometric input to a given object. Colors and other parameters which vary over the surface of an object are often defined as images which are mapped onto the surface as textures. Textures can be generated from photographs or other means, or they may be entirely hand-painted, but most often they are produced by a combination of painting and processing of photographic imagery in an image manipulation tool like Adobe Photoshop. Textures are then mapped onto the surface of a model, and often further manipulated in place, in a real-time 3d paint tool, which provides a simple textured interactive rendering of the model.

Motion: Animation

Once shapes have been modeled, their motion is defined by animation. Although the models they animate may be of high geometric complexity, animators are primarily concerned with the motion of these models. Thus, they commonly work with low-resolution proxies of the fully-detailed high resolution models. They generally need not see the textures or materials of the object being animated, and so can work interactively in simple wireframe or smooth-shaded OpenGL views of their low-resolution proxy models.

Appearance: Shading & Lighting

Finally, the RenderMan-centered production pipeline of the modern special effects or digital animation studio culminates in the work of the technical directors (TDs) creating the final

"look" of each scene through lighting and shader design. The work of the TD mirrors that of the gaffer on a conventional film set, positioning and fine-tuning dozens of lights to perfect the illusion of complex natural or artificial illumination called for by a given scene. Notably — in spite of major research into realistic image synthesis through physical simulation — the TD does not traditionally light a scene with physically meaningful light sources, just as the gaffer would not likely light an "outdoor" scene shot on a soundstage with a singular light in place of the sun. Rather, the TD, like the gaffer, controls scene illumination to artistic effect through the careful placement of many lights throughout the scene; moreover, unconstrained by real-world physics, the TD can further control illumination with special lights which only cast highlights, "bounce" lights which approximate diffuse interreflection between objects, and lights which only affect individual objects.

The TD's work on a scene begins only once the models have been fully constructed, textured, laid out, and animated. The TD lights each shot from one or a few key predetermined camera position(s) which will be used for the shot [Apodaca and Gritz, 2000].

2.3 RenderMan

The artists' work is ultimately output and their images rendered using a technology known as RenderMan. RenderMan is a standardized interface for 3d scene description, first developed by Pixar in the 1980s, and the respected standard for research in realistic computer graphics. Large production studios almost universally use Pixar's PRman — the de facto RenderMan implementation — as their primary renderer. PRman is a scanline renderer built around a REYES rendering pipeline, which, while somewhat different from a conventional z-buffer renderer [1], for our purposes follows the rendering process we have already outlined. PRman implements the RenderMan Interface and Shading Language as the primary mechanisms for scene description [Upstill, 1989; Apodaca and Gritz, 2000]. Film production pipelines and tools are largely built around the RenderMan model of scene description and rendering.

[1] Where a conventional rendering pipeline tessellates, transforms, and rasterizes geometry into fragments which are shaded directly into pixel samples, a REYES pipeline performs shading before rasterization, but it does so on micropolygon grids which are "diced" to sub-pixel size and are functionally treated as point samples, only with shading interpolants computed through tessellation rather than rasterization.

2.3.1 The RenderMan Interface

The foundation of the RenderMan standard is the RenderMan Interface. The interface, commonly controlled through a standard bytestream format (RIB), is analogous to PostScript: it presents a standardized, low-level scene description interface by which the geometric features of a virtual scene — objects, lights, and viewpoint — are described to a renderer, just as PostScript offers a low-level page description interface by which the 2d geometry of a printed page is described to a printer.

2.3.2 The RenderMan Shading Language

In addition to basic geometric features, the RenderMan Interface description of a scene associates shader programs with each object and each light and binds input parameters to each instantiation of these programs. Shaders are subroutines which perform secondary processing during various stages of the rendering process. They are described in the RenderMan Shading Language [Hanrahan and Lawson, 1990].

The shading language is a special-purpose, C-like programming language, orthogonal to the scene description interface and specially designed for the computation of shading and lighting. It includes support for most standard C-style expressions and control flow constructs, including arithmetic and logical operators, as well as loop and conditional constructs. It additionally includes a variety of special-purpose loop-like constructs central to its abstraction of the rendering process. A more detailed introduction to the relevant features of the shading language is presented in appendix A.

2.4 The Lighting Design Problem

Despite the rapid development of interactive computer graphics in recent years, the look design tasks of the TD remain completely non-accelerated and non-interactive. While they work, TDs are forced to wait tens of minutes for feedback because even a single change requires executing the entire final-frame software rendering pipeline over again. Modelers and animators, by contrast, receive instant feedback in an OpenGL-accelerated view port as they finesse control vertex and motion key-frame parameters.

Gershbein and Hanrahan presented the first reasonable approach to interactive lighting design for film production [Gershbein and Hanrahan, 2000]. They observed that lighting design is generally an iterative process of fine-tuning lighting parameters from a single fixed

viewpoint. Therefore, the majority of the computation required to render after each parameter modification is redundant, never varying from one preview to the next. They exploited this by carefully factoring the rendering process, caching common intermediate data in an image-space deep-framebuffer, and only recomputing the final shading of each pixel from the light-dependent portions of these shading computations and the cached intermediate values. They further recognized the effectiveness of hardware acceleration by mapping some final shading operations to the texture and blending functionality of early, non-programmable real-time graphics hardware. Using this approach, they achieved interactive rates with relatively simple lights and shading models in scenes of moderate geometric complexity.

However, in spite of its promise, Gershbein and Hanrahan's deep-framebuffer approach has faced significant challenges in its adoption. Because the approach relies on the use of a small set of predefined materials, it is difficult to apply in production, where RenderMan is used specifically because it allows artists to define complex, custom material and light models through the use of programmable shaders. Established studios and artists possess a large legacy of existing shaders, as well as tools, workflow, and skills invested in RenderMan and the RenderMan Shading Language. The prospect of transforming an entire production pipeline to support an interactive preview tool for lighting has proved unappealing. Even if new material models were programmed in tandem into the lighting design system and into RenderMan shaders, there are significant limitations to the complexity of the materials which can be previewed interactively because Gershbein and Hanrahan's system was not targeted to programmable graphics hardware.

Other commercial work has explored the general problem of efficient re-rendering. Pixar's relatively young Irma tool [Pixar, 2001] professes to accelerate the look design process by caching a large amount of intermediate rendering work in a given frame so that less needs to be recomputed for each small parameter change. This approach is software based and fully general. As a result, it supports arbitrary scenes, but it also very slow and extremely memory-intensive. Also, some commercial 3d packages include general software re-renderers. Maya calls its re-renderer IPR, and 3ds Max calls its ActiveShade. Worley Labs provides two related plug-ins to the Lightwave package — G2 and FPrime — which implement a high-performance preview renderer, and extend this renderer to support fast re-rendering. No such packages exploit programmable graphics hardware, none achieve real-time rates on scenes of even moderate complexity, and none operate on RenderMan scenes.

Because of the clear need to avoid excessive redundant computation during the computationally expensive rendering process, particularly as used during lighting design, prior work has been done in the computational specialization of rendering.

2.5 Computational Specialization

Computational specialization is a compilation technique which employs partial evaluation to increase a program's performance when a portion of the inputs to the program are known in advance. This can be achieved using either "program specialization" [Jones et al., 1993] or "data specialization" [Knoblock and Ruf, 1996]. Program specialization encodes knowledge of the early inputs into a new version of the program, which will usually contain simpler control flow and less computation than the original. However, code size is often a problem since partially computed values are stored directly in the program. Data specialization, by contrast, generates two new programs. The first generates and caches intermediate values which can be computed directly from the prior known inputs. The other uses this cache of precomputed intermediate values to compute the final results for a given assignment to the remaining inputs. Both methods can also be combined in order to leverage the different advantages which both offer [Cutts et al., 1999].

These ideas have been applied to graphics systems before. One early application-specific use was Hanrahan's surface compiler [Hanrahan, 1983]. It produced a portion of a C ray tracing program tailored to the particular algebraic surface being rendered. Partial evaluation was applied more generally to ray tracing by Mogensen [Mogensen, 1986], and this topic was further studied by Andersen [Andersen, 1996]. Guenter et al. applied data specialization to procedural shaders [Guenter et al., 1995]. Their system attempted to solve problems similar to relighting by reducing the amount of time spent shading when only certain parameters were varying. Their system targeted a software renderer using a locally developed shading language that was a syntactic subset of C. It was more general in scope than relighting, since any shading parameter or set of parameters could be varied. They attempted to solve the fully general problem of automatically performing a minimum of recomputation under relatively arbitrary conditions, and faced substantial practical challenges as a result. As is common with lazy evaluation techniques, they suffered a data explosion and reduced special-case performance in their effort to achieve maximum generality.

The technique we present avoids the problems of lazy evaluation by caching data at a

single, fixed stage of the rendering process — during shading and lighting — where data flow and data access are highly regular. We apply specialization directly to shaders, rather than to the entire rendering process. We thus avoid the challenges of specializing C code, with its aliasing and other ambiguities, and we are able to leverage the special-purpose design and data flow model of the RenderMan Shading Language to achieve greater optimization in practice. Finally, because we specifically target lighting design, we can exploit the even more regularized nature of light-dependent computations, in particular.

Chapter 3

Approach

Central to our approach is an interactive rendering engine, specially designed to provide very fast feedback during lighting design while maximizing the fidelity of the preview to the final rendering. The interactive renderer produces output mathematically equivalent to the final renderer, but it achieves interactive performance through three primary optimizations. First, we computationally specialize the rendering process during the shading and lighting stage. Second, we cache the contribution of all static lights, only recomputing the light currently being edited by the designer. Third, we map the resulting specialized and optimized computation to programmable graphics hardware to achieve interactive performance. Through these optimizations, our technique achieves dramatically faster rendering performance than prior attempts to solve the lighting design problem, while simultaneously achieving far greater accuracy than all prior interactive techniques.

Computational Specialization

In the deeply pipelined process of content creation, TDs only modify parameters to the shading portion of the rendering process. Because they generally light a scene from one or a few predetermined viewpoints, the only parameters varying over the course of the lighting design process are the parameters to the shading computations. Thus, all stages of the rendering process prior to shading are static during lighting design. If the full rendering process is employed each time the lighting is previewed, these static stages are redundant, recomputing identical results each time a new preview is rendered. Furthermore, a large portion of even the shading stage is redundant during lighting design. Because TDs generally only modify a subset of the shading parameters which correspond to the parameters of the

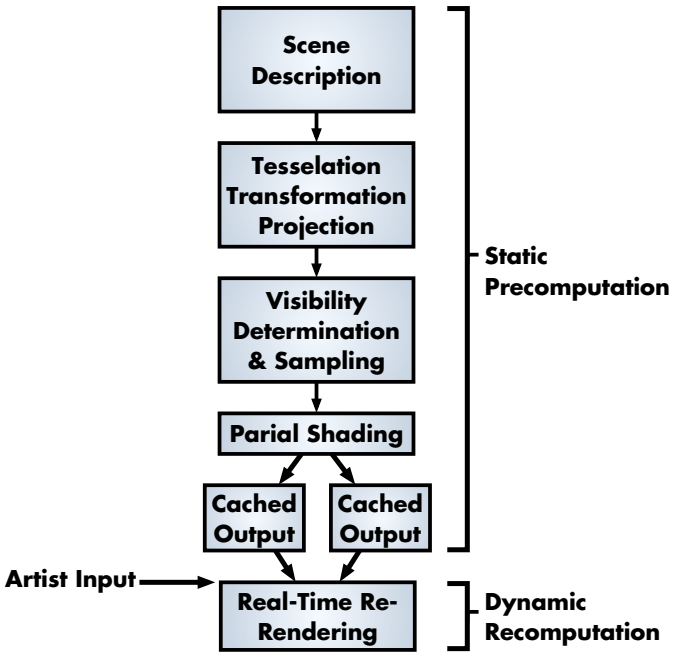

Figure 3.1: Specialization of the rendering process

lights in the scene, much of the shading computation executes unchanged from preview to preview, wasting even more computation.

In order to reduce the computational cost of each preview, we segment the rendering process at the shading stage and computationally specialize the shaders for only the parameters which the user will modify. Caching all static computations and recomputing only those portions of the shading computation which depend on the lighting parameters being edited by the user eliminates all earlier stages of the rendering process. Finally, by rendering only one sample per pixel, each displayed pixel corresponds directly to a single shading sample. Thus, we remove the resampling stage from the rendering process, further simplifying recomputation. Since previews are commonly rendered with little or no antialiasing, this yields identical results to most full preview renders.

Reducing our renderer to a small subset of the shading stage dramatically reduces both the quantity and variety of computation and data necessary for re-rendering lighting previews, significantly accelerating rendering and enabling further aggressive optimizations.

Caching Inactive Lights

TDs commonly light scenes with many independent lights. Rather than allowing pure simulation to achieve realistic illumination from a few physical light sources, they carefully control many small, individual lights to achieve a precise artistic effect. The shading stage of the rendering process is particularly expensive because much of it must be computed repeatedly for each of these many lights, and their contributions combined to yield the final shading for each point. However, during lighting design, a TD will traditionally only change the parameters of one light at a time, since fully configuring each light requires substantial finesse.

Because the radiance visible at a point due to reflection is generally linear in the irradiance onto that point from the light sources, we can consider the contribution of each light in a scene to be linearly independent. We therefore cache the contribution of all lights in their current state, and only recompute the shading for the light currently being manipulated. Combining the dynamically computed contribution of the currently active light with the cached contributions from all other lights in the scene yields the final shading. Thus, rendering performance is made constant relative to the number of lights in a scene, and shading computation is reduced by an order of magnitude or more in common scenes with on the order of 10 lights.

Mapping to Graphics Hardware

The factored computation with which we are left is exactly the type of "embarrassingly parallel" computation which characterizes graphics. Most geometric and procedural texturing computations, which perform irregular data accesses and commonly include data-dependent branching and calls to external C libraries, are entirely independent of the lighting parameters. With the exception of shadow maps, even textures — the largest data structures in most film scenes — are not commonly sampled with any dependence on lighting parameters. Indeed, this is supported by simply considering the division of labor in the production pipeline: separate individuals are responsible for creating texture maps, defining the appearance of surfaces (including the placement of texture maps), defining geometric details, and designing lighting. Each of these individuals separately and independently defines each of these properties of the final rendered image. Therefore, by nature, the parameters edited

at each stage in the pipeline must be independent, the parameters of the lighting not altering the appearance or placement of textures, or the procedural appearance or geometry of objects.

We therefore map the re-rendering computation to programmable graphics processor hardware (GPU), which provides an efficient substrate for the local, arithmetically intense, highly parallel computations which remain from our factorization, and which is already directly connected to the user's display. In contrast to more general parallel systems which rarely approach peak performance, GPUs are optimized specifically for shading computation. Efficient utilization of the dense parallel computation resources of a modern GPU yields an order of magnitude performance improvement over the maximum theoretical throughput of a top-of-the-line CPU, and integration of the re-rendering computation with its real-time display removes the substantial system overhead of an offline software renderer by replacing it with a hardware rendering pipeline specifically designed for interactive rendering.

As a whole, our approach enables smoothly interactive rendering of real-world RenderMan scenes during lighting design, while producing effectively identical results to a full software rendering by automatically specializing and re-rendering from the exact final rendering computations.

Chapter 4

Implementation

We implement our approach in a compiler-driven system which enables automatic specialization and re-rendering of unmodified RenderMan scenes. Our system precomputes a specialized cache of all static computations directly from the input scene. It then re-renders from the static cache and the dynamic lighting parameters in a generalized re-rendering framework which runs interactively on programmable graphics hardware. The shaders which drive these precomputation and re-rendering stages are automatically generated by a specializing compiler for the RenderMan Shading Language.

4.1 Precomputation and Caching

We perform all precomputation in the primary software renderer using the final scene description of a single key frame of the shot being lit. To do so, we leverage built-in functionality of the RenderMan Shading Language.

Cached values are precomputed in the final software renderer by providing as input the original scene description combined with modified versions of the original surface shaders for each surface in the shot. These shaders are modified with the addition of secondary output variables — one for every intermediate computation to be cached prior to re-rendering. Each of these intermediate values is output directly from the point in the shader at which it is computed by assigning the result to the corresponding output variable.

We can further use this facility to identify the shader associated with each pixel. By simply outputting a single shader ID channel for all shaders and assigning to it a distinct numerical identifier for each shader, encoded numerically as a floating point value, we can

clearly identify the shader corresponding to each pixel in the deep-framebuffer.

Since each of these additional values is output once at each shading sample, the renderer outputs each secondary output value to a separate rendered "image" representing the value computed and sampled in image space. Because the cached values are only stored once per pixel in the resulting images, we render this precomputation pass with a shading frequency of one sample per pixel to avoid the ambiguity of multiple distinct shading points being resampled into a single output pixel.

We store the precomputed shading intermediates in a structure commonly referred to as a "deep-framebuffer." This structure mirrors the regularized data flow of interpolant values from the rasterization to the shading stages of the rendering process. It consists of an arbitrary set of shading interpolants and intermediates, sampled in screen space, and stored as channels in an image of the scene. The scalar components of a vector value are stored in the spectral channels of a color image, and scalar values are represented either as monochromatic images or as individual channels in color images. On the GPU these deep-framebuffer channels are represented as floating point textures.

Though different intermediates are stored for each surface shader visible in the shot, because each deep-framebuffer stores only one sample per pixel and so only one shader can exist at each sample, it can associate one intermediate value from each shader with a single channel, rather than requiring one channel for each distinct intermediate value across all shaders in the scene. The size of the deep-framebuffer cache is therefore bounded not by the number of cached values across all shaders, but by the maximum number of intermediate values which must be cached for any one shader.

4.1.1 Advantages

Performing all precomputation directly in the host renderer provides several critical advantages in our goal of maximizing the fidelity of our preview to the final output of the software rendering system, while simultaneously simplifying the construction of our special-purpose software and reducing the maintenance required to support future revisions of the host rendering system.

Simplicity Because all processing of scene description and geometry data occurs in the host renderer, we need only recreate one stage of the rendering process for our interactive re-renderer. We need only consider one type of data entering and flowing through this

stage — a regular array of sampled interpolants and shading intermediates — on which we perform a single set of computations to generate a regular array of shaded samples.

Our one-stage re-renderer need only implement a single interface — the RenderMan Shading Language — rather than both RenderMan interfaces implemented by a full rendering system. Even within the RenderMan Shading language, we only implement the subset of the language and its runtime model relevant to light and surface computation. Specifically, we only implement the conventions, semantics, and constructs of the light and surface shader types, ignoring displacement and other shaders critical to the appearance of the scene, but independent of the light re-rendering we wish to compute.

Accuracy Maintaining perfect synchrony with the features and details of the offline rendering system is critical to the utility of a practical system not only to support major new features, but above all to maintain the high fidelity of lighting previews to the final rendered output produced by the evolving software renderer. Prior systems have failed in practice because failing to perfectly reproduce the intricacies of the software renderer can yield unpredictable results which may differ noticeably from the final rendering, making the previews untrustworthy and therefore useless in the eyes of lighting designers.

Because of the relative simplicity of the rendering subprocesses which we replace, and because we perform all precomputation using the same software path which computes the final rendering, our approach can better and more easily match the exact final render than a preview system which did not integrate directly with the final rendering system.

The geometric primitives and system features available in the RenderMan Interface inherently include numerous subtleties in their exact implementation. However, because we perform precomputation in the same software which will compute the final image, using the same input, our system can forego any knowledge of the intricacies of the RenderMan Interface and its implementation in the final renderer while precisely reproducing their results. Though our subdivision of the rendering process is primarily chosen to maximize interactive performance within the constraints of lighting design, it is particularly advantageous because the precomputation which we perform yields intermediate results which are, by definition, mathematically identical to the corresponding values during the final render.

Furthermore, in contrast to the potential implementation subtleties of the RenderMan Interface, the lighting and shading portion of the Shading Language provides a well-defined

programming language describing the exact logical and arithmetic operations which compute the shaded color at each pixel from the input at that point. As in a proper programming language, any subset of a shader computation is inherently well-defined by the corresponding program. Therefore, since our system performs the same re-rendering computation from the same intermediate values which are used during final rendering, its results are mathematically equivalent to the final software render.

Feature Parity Several prior attempts to implement faster RenderMan preview renderers for lighting design have failed in practice because of feature rot. Over their more than 15 years of public use, the RenderMan Interface and Shading Language have seen many major revisions and feature additions. Revisions frequently add geometric primitives like new surface types, as well as systems-level features such as deferred data access and level-of-detail controls. The shading language has been extended with new built-in functionality and some new constructs such as additional message passing functions and distinct vector and normal types. Such prior systems, which have attempted to support RenderMan at multiple levels, and with relative independence from any single software host renderer, have uniformly failed to withstand the burden of revision and addition to the RenderMan standards and the final renderers with which they are used.

Relying on the host rendering system for precomputation of all rendering processes aside from the light-dependent shading of surfaces allows our system to maintain synchrony with the evolving interfaces and features of the host renderer over time. By only directly operating on a subset of the shading process, we can avoid implementing the vast majority of new functionality with most revisions, only requiring the ability to parse and analyze relevant new shading language constructs.

4.2 Real-Time Re-rendering

On the user's side of our system we implement a generalized framework for interactive re-rendering during lighting design. This framework takes the precomputed deep-framebuffer and specialized GPU re-rendering shaders and uses them to render lighting previews at interactive rates. It exposes all editable lighting parameters to the lighting designer and allows her to modify, add, or delete arbitrary numbers of lights of any type for which a shader has been provided.

The framework automatically loads and analyzes the GPU shaders and deep-framebuffer files which are fed to it as input. It differentiates samples by reading the shader ID channel of the deep-framebuffer, with which it can determine which shader to associate with each sample. For efficiency, it preprocesses the deep-framebuffer into groups of samples associated with each shader. All samples associated with a single shader can then be rendered together using single binding of shader parameters and textures to that shader. Furthermore, rather than rendering samples as individual points, they are grouped along scanlines into spans — one-pixel-high rectangles covering contiguous blocks of samples — requiring less vertex bandwidth to render on the GPU.

Although cached values are accessed in the deep-framebuffer as textures, no explicit texture coordinates are necessary to perform this lookup. Rather, by nature of the deep-framebuffer, the texture coordinates for all cached values at each sample are simply the coordinates at which the sample is being rendered. This simplifies the connection between the framework and the given shaders and deep-framebuffer being rendered, as the shaders, themselves, manage the their own texture coordinates, requiring no additional state management in the rendering framework.

The framework works with GPU shaders represented in a high-level shading language. It is written so as to be relatively agnostic of the actual shading language used, but all current work has used NVIDIA's Cg language and libraries [Mark et al., 2003]. It automatically analyzes the shaders to extract all user-configurable lighting parameters and their types, and uses this information to expose sets of configurable parameters for the currently active light, allowing automatic construction of user interface controls for arbitrary light shaders or programmatic control of light shader parameters.

The use of a high-level GPU shading language helps support this automatic support for new light types, while also affording implementation independence from any specific hardware and supporting unique performance optimization on different present and future GPU architectures.

The re-rendering framework is self-contained, respecting the OpenGL transformation stack and other conventions, and does not require ownership of the context into which it is rendering. This allows any OpenGL application to use the framework in any number of ways by invoking its render method within any active OpenGL context, either in a special-purpose lighting application or directly in the viewport of an existing modeling package such as Maya. The framework was specifically designed with the intention of a host application

drawing its choice of lighting or other widgets directly atop or within the lighting preview, or using its output in subsequent real-time compositing operations which produce a final preview.

4.3 Automatic Shader Specialization

Precomputation and real-time re-rendering are performed directly from a standard RenderMan scene by computational specialization of the shaders in that scene. We perform shader specialization automatically using a novel RenderMan Shading Language compiler. This compiler takes as input conventional RenderMan Shading Language shaders and processes them in three stages. First, it performs a global static/dynamic data flow analysis to determine which data and computations are dynamic with respect to the parameters being edited by the lighting designer. Next, it specializes the input shader using this analysis, transforming the computation into a precomputation pass and a re-rendering pass. Finally, it performs code generation on each of these two passes, generating a RenderMan Shading Language description of the precomputation and a corresponding GPU shading language description of the re-rendering computation.

4.3.1 Analysis

Before transforming the shader computation into its specialized form, the compiler must first analyze the shader to determine which computations may be cached during re-rendering. It computes the static/dynamic state of each operation in the shading computation using a fast yet precise two-pass analysis of the shader code.

In order to achieve a minimal overestimate of the global flow of dynamic data through the shader program, the analysis is formulated as a context-free reachability problem. This technique has been developed in the compiler community as a general technique for efficient and precise global data flow analysis [Reps et al., 1995].

Context-free reachability is an extended form of the graph reachability problem wherein edges are not only directional, but may be classified as open/close pairs. As in conventional reachability, the least upper bound of a set is expressed by connecting a so-called “subtype” edge — a simple directed edge — from each member of the set to the dependent node. In the context-free problem, reachability paths across open and close edges are considered legitimate only if the paths contain matching, nested pairs of open/close edges. Individual

open and close edges may be tagged with unique identifiers, and open edges with a given identifier only match close edges labeled with the corresponding identifier.

Such a formulation is critical to efficiently achieving a precise analysis of the dynamic nature of the program flow across function calls. Multiple invocations of the same function require consideration of polymorphic recursion. That is, the recursive analysis of each call to the same function differs based on the specific instantiation of that function — in our case, the static/dynamic nature of the arguments at each specific call site. A context-free analysis of reachability through these sites can effectively group all arguments to an individual function invocation and precisely analyze their flow through the function and back to the specific return site.

This formulation is actually more general than most practical RenderMan Shading Language implementations, since it transparently supports recursive function definitions, which, while not disallowed by the RenderMan specification, are not used in practice because Pixar's RenderMan implementation does not support them. However, this formulation is not chosen with the intention of creating excessive generality in the analysis — rather, it offers the most general, scalable, and well-understood formulation which is able to achieve a high-precision global flow analysis. Furthermore, it renders this technique useful even in view of the possibility of support for recursion in future standard implementations, and it presents the opportunity to potentially generalize these analyses to other C-based shader languages, such as that used by mental ray, as well as potentially supporting analysis through custom C-language shader operations used in RenderMan shaders.

Data Flow Analyses

The analysis begins by initializing each input to the shader to its initial static/dynamic state. A special root node is created to correspond to the value *dynamic*. This node is then "assigned" to each dynamic input by creating a simple subtype edge from *dynamic* to each dynamic input variable. Because we are specifically analyzing for light-dependence, we initialize L and Cl to dynamic, as well as all parameters to any light shader. Furthermore, whenever a variable is assigned by the lightsource() message-passing call, it is directly assigned the value *dynamic*. In this formulation, all nodes not reachable from *dynamic* are implicitly deemed static.

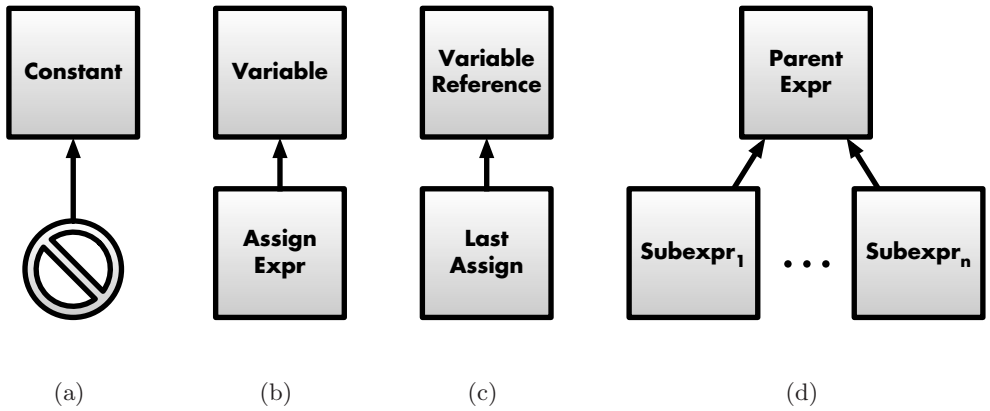

Figure 4.1: Reachability formulation of data flow analyses for the four base expression types: (a) constants, (b) variable assignments, (c) variable references, and (d) compound expressions.

Constants Constant values are static under all possible conditions. They implicitly receive this value by inheriting from no other nodes in the program.

Variable assignment Variable assignment overwrites the static/dynamic condition of the variable to which the assignment is made with the expression that is assigned to it.

Variable reference Variable references inherit along a subtype edge from the last prior assignment to the variable, stored in a scoped variable table. This flow from assignments to uses connects the data flow between statements in the shader.

Operators and other local expressions All local expressions and constructs, such as arithmetic, relational and logical operators, type casts, and statement blocks, inherit their static/dynamic status according to the least upper bound over their children. In the graph formulation, a subtype edge connects from each child to the parent expression.

Conditionals `if` statements, and corresponding ?:-style conditional expressions, create branches in the data flow. Conditionals are dynamic according to the least upper bound of the condition and the statement. That is: dynamic body statements must be executed in order to fully compute the dynamic results, while dynamic conditions must be computed in

order to determine whether or not the effects of the potentially static statement body take effect after the conditional. Thus, a conditional must be considered dynamic if either its condition or its statement is dynamic. In this respect, the conditional, itself, is local.

However, data flow through conditionals involves branches and is therefore global. The static/dynamic assignment to any variable assigned in the body of the condition statement may be determined by the last assignment to that variable within the conditional, if it is executed, or by the last assignment prior to the conditional, if it is not. Therefore, immediately after a conditional, the variable depends simultaneously on the last assignment prior to the conditional, the last assignment within the conditional, and the condition on which the conditional is predicated, which will determine which assignment takes effect.

This behavior is scoped within the bounds of conditionally executed blocks. While variable assignments at the same scope explicitly overwrite one another, variable assignments in nested scopes are pushed onto a stack. At the end of a conditional scope, this stack is popped and new values for each variable assigned within the conditional scope are created according to the above rule.

In the case of if/else conditional pairs, this global effect is reduced: any variable which is assigned in both the if and the else statements unconditionally overrides any possible prior assignment.

Loops for and while loops behave like an extension of conventional conditionals. The statements over which a loop iterates execute conditionally on the bounds of the loop. However, loops have the added property that data flow exits from the loop statements not only to the beginning of the next subsequent expression after the loop, but also to the beginning of the loop condition and body. Furthermore, loops support continue and break statements which connect flow directly to the beginning of the loop and the statement after the loop, respectively. Thus, analysis of the loop must connect the scope from any possible continue point in the loop — the end of its body or any explicit continue expression — to the beginning of the loop, and from any break point in the loop — the end of its body or any explicit break expression — to the statement following the loop. The flow from break points to the statement following the loop mirrors the flow across conditionals. Capturing the flow through continue points requires iteratively re-visiting the body statements after popping the conditional scope at each potential continue point.

Integration Loops Like conventional loops, the illuminance, illuminate and solar constructs execute their body statements conditionally based on implicit or specified loop bounds. Therefore, flow through integration loops is similar to that through conventional loops: the construct must be deemed dynamic if either its statement or bounds are dynamic. In each of these constructs, the statement and the bounds are connected to the loop construct by subtype edges.

Function calls Data flow through function calls is both global and polymorphic. Because functions may be invoked with arbitrary assignments of static/dynamic arguments at any individual call site, flow through function calls must address polymorphic recursion in order to achieve a precise estimate of flow through any individual function call. Each call site is identified with a unique index. This index is then used to label the context free open/close edges which connect flow through the function. Open edges are created with this index from each argument expression to the argument variables of the function implementation, and a corresponding close edge is created from the return site to the result of the function call. The RenderMan Shading Language specifies that all non-void functions provide one and only one return site, simplifying this analysis.

Built-in functions require special handling, as they are not represented in the source code of the shader. The analysis engine encodes specific knowledge of their implementation and directly inserts the appropriate data flow at each call site. This is of particular importance for built-in functions which access global data beyond the that which is passed in as arguments. The illumination functions diffuse(), specular() and phong() execute illuminance loops in the scope of the caller, and their results thus depend on L and Cl. In most other cases, the result of the function simply depends on the least upper bound of all the arguments passed at that invocation.

DSO shadeops are assumed to depend exclusively on the arguments passed to them, and are therefore analyzed like opaque built-in functions.

Message passing routines have relatively unique semantics, and must be analyzed as such. These routines receive their primary "return" value through a pass-by-reference argument which is evaluated for type at runtime and conditionally assigned a value according to the semantics of the given routine. A simple analysis treats all message passing calls as static, except for those receiving dynamic arguments or those accessing dynamic values. In

our application, only lightsource() routines accessing values of the lights marked as user-configurable are deemed dynamic. The static/dynamic assignment of the message passing call is assigned directly to the pass-by-reference result variable.

Once it has fully analyzed the shading computation, the compiler now has a complete annotation of the static/dynamic nature at every point in the computation.

4.3.2 Specialization

Using the static/dynamic information from the analysis stage, the specialization process operates directly on the abstract syntax tree (AST) from the input shader. Prior to any transformation of the computation, the AST must be duplicated into separate precomputation and re-rendering versions. The specializer then constructs output nodes where necessary in the precomputation AST and replaces these outputs with cache lookup operations in the re-rendering AST.

Transformations are performed at the transitions between static and dynamic portions of the source shader. These transitions occur where dynamic expressions include static subexpressions.

Precomputation

The precomputation pass performs the entire shading computation as defined in the source shader, but it is extended to output all static subexpressions of the dynamic computations in the shader. Because the C-like RenderMan Shading Language supports assignment operations as subexpressions which evaluate to the value assigned, the precomputation pass can be transformed from the input shader by simply replacing all static subexpressions of dynamic parent expressions with a parenthesized assignment expression which assigns the static value to an output variable. Because this new assignment operation will evaluate to the same value as the static subexpression, itself, this allows the static subexpressions to be evaluated in the exact context in which they are computed in the source shader, while enabling the shader to continue execution past this point of output to compute further static subexpressions.

Boolean subexpressions can be cached using a subtle variant of this method by which they are replaced with an assignment from a conditional expression yielding the floating point value 1.0 if the condition evaluates to true, or 0.0 if it evaluates to false.

Re-rendering

Mirroring the precomputation pass, the re-rendering pass performs only the dynamic portions of the original shading computation, using as inputs the cached static subexpressions from the precomputation shader. The input shader computation can be transformed into the re-rendering shader by deleting all static statements and replacing all static subexpressions of dynamic expressions with a lookup into the corresponding channel of the deep-framebuffer cache. This cache lookup is simply a texture access to the corresponding deep-framebuffer channel at the current deep-framebuffer coordinates.

Because the precomputation and re-rendering transformations each perform parallel traversals of the copies of the same abstract syntax tree, the deep-framebuffer channel assigned to each static subexpression need not be communicated between the two processes. Since the expressions will be traversed in the same order, the appropriate deep-framebuffer channel can be easily inferred independently in either process.

Mirroring the caching of static boolean subexpressions as binary floating point values, cached boolean expressions, stored as binary floating point values, can be replaced with a complementary conditional which compares the cached value to 0.0 to determine its truth.

With these transformations, the compiler specializes the original computation of the input shader into separate static precomputation and dynamic recomputation programs, which together perform the same computation and produce the same results as the original shader executed in whole.

4.3.3 Code Generation

After the specializer transforms the input shader into complementary precomputation and re-rendering computations, the backend of the compiler generates corresponding shader files which perform the complementary stages of the specialized computation.

RenderMan Precomputation Shader

The precomputation shader is output as RenderMan Shading Language source defining a version of the source shader modified as described to output static subexpressions into a deep-framebuffer cache.

Because the compiler operates on a RenderMan Shading Language abstract syntax tree, the specialized AST clearly contains sufficient information to output RSL source code.

Code generation proceeds in a relatively straightforward process complementary to the parser which generates the AST, outputting the corresponding RSL code to represent each syntax tree construct. Because typechecking generates explicit type cast nodes where type conversion was specified implicitly in the source shader, the resulting output can include many more explicit type casts than the source shader, but it is computationally equivalent.

Cg Re-rendering Shader

The re-rendering shader is output in the Cg high-level shading language for GPUs.

By virtue of the intentional high-level similarity between the languages, and by the nature of the computation which we are generating, the Cg codegen is fundamentally similar to the RSL codegen. Indeed, recent work has demonstrated the feasibility of directly compiling a large subset of the RenderMan Shading Language to programmable graphics processors [Bleiweiss and Preetham, 2003]. Targeting a high-level shading language with similar C-like semantics simply makes the compilation process simpler and more transparent.

Because both are similarly C-like languages with comparable mathematical functionality, most RenderMan expressions map directly to comparable Cg expressions. Similarly, many more advanced constructs in RSL have direct corollaries in Cg. Thus, most relevant structures in the RenderMan AST are trivial to map to their Cg equivalents. Mathematical operations either map directly to equivalent arithmetic operators in Cg or to standard library calls in the language. Function calls have identical semantics in the two languages, and both exclusively inline all functions in their current implementations, creating identical limitations in practice.

Many standard library functions in the RenderMan Shading Language, such as pow(), abs(), etc., map directly to identical Cg library functions. Those standard library functions which do not map directly to Cg equivalents must be implemented in an extended Cg library. For most such functions, this is a simple matter of implementing them in Cg according to the RenderMan specification. The color functions and most math and geometric functions are straightforward to implement in Cg.

Map Access Functions Map access functions, however, have different semantics in RSL and Cg. In RSL, maps are referenced by name using strings, while in Cg they are identified by texture handles stored in "samplers." So long as the strings assigned to these identifier

variables are treated as opaque objects and not modified through string processing, they can be mapped directly to corresponding texture handles. Furthermore, the RenderMan map access functions accept variable sets of texture coordinates, allowing up to 8 coordinates defining a quadrilateral over which the sample is filtered. This can be approximated by a filtered access on the GPU configured to filter over an area approximately corresponding to the given rectangle.

The RSL map functions further accept an optional, variable length parameter list, consisting of token/value pairs which identify and assign additional, optional arguments to the map access functions. These additional parameters may be addressed by polymorphic implementations of the map functions in Cg which endeavor to emulate the behavior of the corresponding set of optional arguments, but they are a complexity beyond the current scope of this work, as they often provide only configurable qualitative improvements in map sampling, and often do not radically alter the values accessed from the texture. Above all, texture accesses are far less common in the light-dependent re-rendering computation than in a complete shader, and therefore, while necessary for reasonable generality, require less special attention than functions more central to common lighting computation.

Specialized map access functions bump() and shadow() have additional high-level functionality in RenderMan, beyond simply looking up a value in the corresponding map. This functionality can be emulated using Cg functions which wrap around the raw texture lookup and perform the same specialized computation to abstract the map contents as a bump or shadow map. Shadows, in particular, are addressed later in more detail.

Coordinate Space Transformation Some geometric functions require additional information to implement on the GPU. Of particular interest, the transform() function allows transformation of vector types between different coordinate spaces defined in the current shader context. Like the map access functions, transform() accepts coordinate space identifiers as string names, but, if static, these names can be mapped to corresponding numeric variables on the GPU. These coordinate spaces are challenging to handle because many of them vary over the surfaces being shaded. The GPU has no way of deriving the appropriate coordinate transformations for each sample and therefore cannot easily evaluate them in-place during re-rendering. However, the compiler can employ the output variable functionality of the RenderMan Shading Language to compute a per-sample varying transformation matrix for each transformation which must be applied on the GPU. Wherever

a given transformation is used, the precomputation shader can store a set of basis vectors for the given transformation by computing a corresponding transform() on the vectors (1,0,0), (0,1,0), and (0,0,1) and storing the result. The equivalent transformation can then be computed on the GPU from the basis vectors cached at each sample.

The Data Flow Model and Integration The RenderMan data flow model implicitly binds global variables storing the shading interpolants at each sample and capturing the final shaded results. Though these global variable semantics are specific to the different classes of RenderMan shaders, they behave as per-sample arguments and return values to and from the shader entry point. Thus, they can simply be mapped to per-sample varying arguments and return values in Cg.

Further, the RSL data flow model separates surface and light shaders, allowing them to interact via illuminance and illuminate/solar constructs. In the general case, these constructs have no clear mapping to the GPU where all surface and light shading calculation is performed together in a single fragment processor program. However, since only one light is being varied, a simplification can be made. Because only the currently changing light is re-rendered, the contribution of all other lights is fixed. Thus, the abstraction of the illuminance and illuminate constructs is unnecessary. Instead, the light and surface calculations are directly performed in the same shader. The light shader is generated as a function returning the colored light intensity *Cl* at the point currently being shaded, and the surface shader is generated as a similar function, modified to accept this singular value of Cl from the currently active light.

Because of the well-defined, loose binding between surface and light shaders, they can still be compiled independently. However, they generate Cg functions rather than RenderMan shader bytecode. These Cg functions can then be combinatorially combined into Cg pixel shaders for all pairs of surface shaders and light shaders. The resulting wrapper shader accepts the union of all configurable parameters and deep-framebuffer channels input to the surface/light function pair. It then simply binds each of its arguments to the corresponding light and surface parameters, invokes the light function to compute Cl for the current point, and invokes the surface function with the computed Cl to produce the shaded color. A simple script can perform this surface/light combination as a post-compilation process. The same effect could be achieved using Cg's support for abstract method interfaces.

Chapter 5

Challenges

Not all RenderMan Shading Language functionality is feasible to support with full generality on contemporary graphics processors.

Some math functions, such as mod() and the corresponding % operator, as well as any bitwise operations, have no clear implementation in current GPU instruction sets. Similarly, some more complex functionality such as string processing, many message passing functions, and custom shading operations implemented in external C DSOs, have no feasible or meaningful GPU analog. Ray tracing functions are extremely expensive and require data and processing outside the scope of our deep-framebuffer data structure. These features are not currently handled in the Cg codegen. In practice, however, this does not generally limit their use in shaders. Rather, our results demonstrate that most such features and constructs which have no clear GPU mapping tend to occur little or not at all in the light-dependent portion of the shading computation, even in complicated shaders which make aggressive use of these features.

5.1 Data-dependent Loops & Conditionals

Conditionals and loops are supported in the Cg language, and can therefore be expressed easily by the Cg codegen, but most current GPU implementations do not implement branching in the pixel shader. Therefore, these language constructs are not directly supported by most present GPU implementations.

However, all contemporary GPUs can emulate conditionals with reduced efficiency by unconditionally evaluating both branches over all samples and using a SIMD write mask

which only outputs the results from the branch which for which the conditional evaluates to true. This still allows some support for conditional computation, albeit less efficiently than with a true conditional implementation in hardware. Furthermore, in real-world RenderMan shaders, many loops and conditionals are not data-dependent, and therefore can be evaluated or unrolled statically. We evaluate the frequency of general and data-dependent loops and conditionals in our results. Finally, recent improvements provided in the PixelShader 3.0 model [Dietrich, 2004] provide direct support for dynamic branching functionality on the most recent GPUs, enabling direct execution of dynamic conditionals and loops on the GPU.

5.2 Illuminance & Illuminate Constructs

We have already addressed the fundamental simplification of illuminance and illuminate in the rendering data flow of the real-time re-renderer. However, a few intricacies must be addressed to fully support the functionality of these constructs.

These constructs implicitly perform loops over all light or surface points interacting with the complementary surface or light point currently being shaded. While the actual iteration of the loops is implicit in the execution of the shader, the integration bounds are specified geometrically by a cone. Though many common uses specify this cone to simply be the hemisphere over the point being shaded or in the direction of the light, the cone can be defined arbitrarily in terms of a position, orientation, and angle. Thus, achieving fully correct results requires evaluating the cones of integration and only accounting for results from interactions which fall within these cones. This bound can be implemented efficiently in the re-renderer by masking the results of the corresponding integration loop based on evaluating the dot product between the between the orientation vector and the vector from the position of the cone to the position of the complementary surface or light point being shaded. This dot product can then be compared to a threshold equal to the cosine of the cone angle. In the common case where these bounds are defined statically, the compiler can evaluate this cosine at compile-time, replacing it with a floating point constant.

Additionally, both constructs implicitly bind additional, per-light and per-point variables for the purposes of integration. These variables are inherently available in the scope of the surface or light shader. However, in common usage, illuminance and illuminate loops

are often implemented inside of decomposed functions, where these variables must be explicitly declared external. When executed in a RenderMan renderer, these integration variables are implicitly accessible where appropriate — the explicit extern declarations are necessary only for proper compile-time type checking. The real-time re-renderer, however, has no such implicit support for RenderMan's integration variable semantics. It therefore must account for all such external variables by extending the prototypes of relevant functions to explicitly pass them down function call chains.

5.3 Derivatives

High quality antialiasing, the calculation of normal vectors, and other common graphics operations require the computation of derivatives to account for the relative rate of change of interpolants and shading intermediates. The RenderMan Shading Language facilitates computing derivatives for arbitrary expressions with respect to the parameters of the current surface or with respect to other expressions. Practical implementations simply approximate the derivative using finite differencing between adjacent samples.

Currently shipping graphics processor architectures, however, perform all shading computation entirely locally to each sample, and therefore do not support the computation of arbitrary derivatives. A Cg shader could manually emulate this functionality by fully evaluating the desired expression at the current sample and its neighbors — accessible by sampling the necessary deep-framebuffer values at its neighboring texture coordinates — and using finite differencing to compute the derivative. This would require redundant computation of neighboring values in at each sample, and could yield errors at surface boundaries. However, further recent improvements in the PixelShader 3.0 model [Dietrich, 2004] include the direct and efficient evaluation of screen-space derivatives for arbitrary functions on the GPU. Derivatives may then be computed relative to arbitrary expressions as the quotient of the screen-space derivatives of the two expressions.

5.4 Message passing

RenderMan's message passing facilities allow shaders to violate the data flow model of the shading computation by sharing data between interacting shaders of different types, and directly between shaders and the renderer. Because the interactive re-renderer operates

independent of all stages of the rendering data flow outside the light-dependent portion of the surface and light shaders, much of this information is not dynamically accessible at runtime during re-rendering. However, by nature of the division of labor in the production pipeline, light parameters are not likely to affect most computation outside of the surface and light shaders. Thus, the results of most other message passing routines will likely be static during re-rendering.

However, two message passing interfaces can be critical to the light-dependent re-rendering computation. lightsource() and surface() allow surface and light shaders acting on the same sample to access each other's parameters by name.

This introduces two challenges. First, parameters are named by strings which are matched at runtime. However, in practice, most usage employs constant string tokens which can be analyzed at compile time. Second, this introduces dependencies between the surface and light shaders beyond the *Cl* value computed for the current iteration of the illuminance loop on the surface by the current light shader. Because the re-renderer relies on the unidirectional data flow from the light to the surface shader at each sample in order to combine the surface and light shaders on the GPU, this raises concern for possible bidirectional dependencies. However, all of the shader-to-shader message passing mechanisms, including lightsource() and surface(), only allow access to the input parameters of the corresponding shader — not to arbitrary intermediate values computed during the shader's execution. Thus, since light and surface components of each re-rendering shader are already combined within a single Cg shader receiving all parameters to both the surface and light shaders, emulating the functionality of the lightsource() and surface() message passing mechanisms is simply a matter of passing the corresponding parameters into additional arguments to the surface and light re-rendering functions and replacing all message passing calls with direct accesses to the newly available arguments.

5.5 Noise

High quality noise is critical to the computation of many computer graphics effects and is therefore an often-used feature of the RenderMan Shading Language. GPU shading languages, however, provide no support for the type of temporally-coherent, multi-dimensional noise functions required for advanced shading. Our evaluation indicates that most specialized re-rendering computation does not perform noise computations. However, recent work

has demonstrated the feasibility of real-time noise computations on programmable graphics processors at varying levels of performance [Hart, 2001] [Perlin, 2004].

Chapter 6

Results & Analysis

Experimental evaluation demonstrates that our technique achieves very high performance. On contemporary graphics hardware, it achieves smooth real-time frame rates for all tested scenes, representing a performance improvement of more than three orders of magnitude over unoptimized software rendering. In order to better understand the performance advantages of our technique, we decompose its performance improvement into three individual optimizations: surface shader specialization, caching of static lights, and parallel computation on programmable graphics hardware.

Furthermore, our analysis demonstrates that the light-dependent dynamic subset of most realistic shaders requires only a limited class of computation which can be efficiently mapped to current programmable graphics hardware. We study this effect of our analysis by profiling the computation of a variety of complex, real-world shaders containing RenderMan Shading Language constructs which cannot be mapped to the GPU.

6.1 Performance

Methodology

We analyze the performance improvements provided by each optimization in our technique by measuring scene rendering performance in the software renderer after applying each successive optimization, and finally we compare a fully specialized software render against our graphics hardware-based real-time implementation. All measurement is performed on a 2.6ghz Pentium 4 workstation with 1gb RAM and an NVIDIA GeForceFX 5900 Ultra

(a) Original software rendering

(b) Real-time preview with added blue rim-light

Figure 6.1: Offline and real-time renderings of the Warrior Bug

graphics accelerator. All software is run under Windows XP, and PRman 11.3.2 is used for all software rendering.

Our test scene is a RIB of the warrior bug from *Starship Troopers 2*, placed above and in front of a ground and back plane. All rendering is performed with shadows disabled. The scene is of low-but-realistic detail for a special effects workflow. It corresponds to a shot in which a lighting designer independently lights a single object which will later be composited into a larger scene. To mirror a realistic workload for such a scene, the bug is lit with 7 lights, including a spotlight key light, a directional fill light, and five additional point lights providing secondary and ambient illumination.

All renderings of the scene are performed at a resolution of 768x768 pixels.

Baseline Performance

The software renderer achieves a baseline performance of 28.7 seconds rendering the full scene with all 7 lights active. This cost is dominated both by the relative geometric complexity of the model, and by the complicated procedural displacement algorithms run over

Preview Performance	**Rendering Cost (sec)**	**Incremental Performance**	**Cumulative Performance**
Baseline (7 lights)	28.7	–	–
Specialized (7 lights)	6.1	4.7x	4.7x
Specialized (1 light)	3.5	1.75x	8.2x
GPU	0.025 (40fps)	140x	1148x

Table 6.1: Warrior bug rendering performance improvements from three optimizations.

this geometry. Displacement shaders programmatically compute much of the finer surface detail over the bug, and include substantial computation to automatically place realistic wounds on the bug's various body parts. These wounds, computed during displacement, interact with the surface shader via message passing. The surface shader uses the initial computations of the displacement shader to procedurally color the wounds and shell of the bug. It then employs a layered illumination model to compute the bug's shading. It represents the bug's shell as a plastic-like semi-glossy surface covered with a thin, non-glossy film of dust. The shell and dust layers are shaded independently and blended according to a dust attenuation map.

Specialized Shading Computation

Employing an optimized shader, the software renderer is able to compute light-dependent shading from a deep frame buffer cache in 6.1 seconds when rendering the scene with 7 active lights.

The same optimized shader requires 3.5 seconds to render from a single light combined with the cached illumination due to the remaining 6 lights.

These software renders are executed much like our real-time renderer. All interpolants and shading intermediates are accessed from texture maps, while editable light parameters are passed directly to the surface shader. Because PRman is not optimized for such use, the basic overhead of the system substantially inflates the cost of rendering. Simply rendering a constant shader over a 768x768-pixel polygon requires more than 1 second to render. Factoring out this minimum system overhead, the performance improvement achieved by caching all inactive lights surpasses 2x. Additionally, the single light shader adds substantial overhead by sampling cached illumination from an additional texture which must be fetched at every sample. Such overhead is exacerbated by PRman's slow texture fetch performance as compared to the GPU which is heavily optimized for streaming texture access.

GPU Re-rendering

Employing all optimizations of our technique, our GPU-based implementation renders the scene in 0.025 seconds, sustaining an interactive re-rendering rate of 40 frames per second. This provides a performance improvement of three orders of magnitude over the baseline software rendering cost, enabling the scene to be rendered at smooth real-time rates, even at the relatively high resolution of 768x768 pixels.

The GPU-based re-rendering pipeline offers far and away the greatest single achievable performance improvement. A large part of this disproportionate gain is likely due to the high overhead of the software renderer, in contrast to the near-zero per-frame overhead of the GPU rendering pipeline. Yet, even if the software rendering pipeline were heavily optimized for re-rendering workloads and fast shading, it would not likely reduce most of the 140x performance delta between the GPU renderer and the most specialized software rendered scene.

However, this disproportionate result does not discount the value of specialization. Rather, while high system overhead may prevent shader specialization from improving the performance of the software rendering pipeline proportionally to the reduction in shading computation, specialization is critical to enabling efficient GPU-based re-rendering in most scenes. Because programmable graphics processors cannot execute a large subset of RenderMan Shading Language functionality commonly employed in production shaders, specialization is necessary to reduce the shader computation to a subset which tends to depend far less on complex RSL functionality: the subset necessary to compute the illumination. Furthermore, the computational optimization more directly improves the effective rendering rate within the low-overhead shader execution environment of the GPU than in the high-overhead software renderer. Finally, specialization reduces the rendering data flow to a single stage which performs local processing on a single, regular data structure of moderate size. This reduction is far more critical to achieving peak performance on the highly parallel GPU architecture than it is to improving the performance of the software renderer.

6.2 Realistic Shader Computation Profiles

We evaluate the effectiveness of our data flow analyses and analyze the feasibility of efficiently mapping the resulting specialized re-rendering computations to existing GPU hardware by studying the computational profile of a variety of interesting RenderMan shaders.

The shaders tested are chosen include both publicly available and real-world special effects shaders. They are selected to demonstrate a variety of advanced shading computations representative of realistically challenging production shaders. They all employ RSL features which cannot be mapped efficiently to current programmable graphics hardware.

6.2.1 Methodology

These shader profiles represent a compiler-based static analysis of the computation specified by each shader. This analysis counts the number of instructions required to implement each block of the shader. It accounts for the cost of function calls separately at each call site, and it accounts for the cost of calls to standard library functions according to simple implementations of these functions described in or inferred from the RenderMan Shading Language specification.

Because it is a static analysis, these profiles cannot reflect the iterations of loops or the conditional execution of if statements. However, although there may be cases in which the execution of branches would substantially alter the relative effects of our shader specialization, our results reveal that disproportionately more loops tend to exist outside the specialized computation than within it. Thus, dynamic analysis would likely reveal a *greater* disparity between the specialized and non-specialized shader profiles.

Three profiles are generated for each shader. The first, "Complete," denotes the computational profile of the entire shader. The second, "Varying," represents the computation dependent on all parameters to the shading system. This specialization is computed using our precise data flow algorithms, and is considered the greatest possible reduction which could be achieved by naive optimization. The third profile, "Dynamic," represents the computational profile of the fully specialized light-dependent portion of the shader.

The computational profile for each shader breaks down its instruction count by type. Arithmetic denotes the total number of individual arithmetic operations computed by the shader. Any arithmetic instructions which cannot be mapped to the GPU are indicated by the "unhandled" subset of the arithmetic instructions. All map access operations are accounted as "texture" operations. Conditionals counts the total number of if statements and conditional expressions, while its "data-dependent" subset includes only those conditionals predicated on a dynamic condition. Loops similarly counts the total number of while and for loops, while its "data-dependent" subset indicates the number of loops with dynamic bounds. All invocations of message passing routines are recorded and broken down

Shader Complexity	Complete	Varying	Dynamic
Instruction Count	**410**	**409**	**23**
Arithmetic	**404**	**403**	**23**
Unhandled	1	1	0
Texture	**0**	**0**	**0**
Conditionals	**6**	**6**	**0**
Data-dependent	6	6	0
Loops	**0**	**0**	**0**
Message Passing	**0**	**0**	**0**
DSO shadeops	**0**	**0**	**0**

Table 6.2: Shader computation profile for the Parquet Plank surface shader.

by the type of message sent. Finally, all calls to external C libraries are accounted as DSO shadeops.

6.2.2 Parquet Plank Surface

Parquet plank is a commonly used example of fully procedural shading in RenderMan. It employs Perlin noise-based wood to shade a surface as though it were a parquet floor. It is chosen for the high complexity of its procedural texturing.

Our analysis reduces the computational complexity of the parquet plank shader by nearly a factor of 20. It eliminates all conditionals, as well as all instructions which cannot be mapped efficiently to the GPU. The naive varying analysis, by contrast, achieves no effective reduction of the shader along any axes.

Approximately 95% of this shader implements the procedural texturing algorithms which compute the parquet floor pattern. These procedural algorithms rely heavily on very expensive noise() routines to simulate the appearance of the wood within the individual planks. The varying analysis is unable to reduce these computations because they are heavily parameterized by the shader's instance variables.

However, all of this procedural texturing code operates entirely independently of the illumination of the surface. For illumination, the shader implements little more than a simple diffuse()+specular() shading model, augmented by the properties of the wood computed by the procedural texturing algorithms. Our analysis is able to fully exploit the complete independence of this simple illumination model from the complex procedural texturing routines.

Shader Complexity	Complete	Varying	Dynamic
Instruction Count	**5364**	**577**	**175**
Arithmetic	**5152**	**527**	**172**
Unhandled	30	20	0
Texture	**0**	**0**	**0**
Conditionals	**155**	**20**	**2**
Data-dependent	155	20	0
Loops	**28**	**20**	**2**
Data-dependent	28	1	0
Message Passing	**29**	**29**	**0**
displacement	29	29	0
DSO shadeops	**0**	**0**	**0**

Table 6.3: Shader computation profile for the Cloud surface shader.

6.2.3 Cloud Surface

The cloud shader is a highly complex procedural cloud shader used heavily in the production of both sequels to *The Matrix* by Tippett Studio. It computes a fully shaded cloud or atmospheric appearance, simulating clouds as seen from the outside. It is entirely procedural, generating the full cloud structure computationally using a novel method (to be presented at SIGGRAPH 2004).

Our analysis reduces the computational complexity of the cloud shader by more than a factor of 30. It eliminates 99% of the conditionals, and the two which remain are non-data-dependent. It similarly reduces number of loops to two, both of which are non-data-dependent. It eliminates all message passing routines, as well as all instructions which cannot be mapped efficiently to the GPU. The naive varying analysis achieves a substantial reduction in the arithmetic complexity of the shader, but cannot eliminate most instructions which cannot be mapped to the GPU.

The cloud shader, like the parquet plank shader, primarily performs procedural texturing computations of extremely high computational complexity. Its illumination is more closely integrated into the procedural appearance of the surface than is the illumination model of the parquet plank shader, and its reflection model is substantially more complex. Our analysis is still able to properly distinguish the texturing portions of the shader from the illumination portions, reducing the shader to a feasible size for real-time re-rendering.

Shader Complexity	Complete	Varying	Dynamic
Instruction Count	**652**	**273**	**99**
Arithmetic	**559**	**227**	**79**
Unhandled	9	8	6
Texture	**3**	**3**	**0**
Conditionals	**70**	**33**	**13**
Data-dependent	70	23	3
Loops	**9**	**0**	**0**
Data-dependent	9	0	0
Message Passing	**9**	**8**	**7**
lightsource	8	7	7
displacement	1	1	0
DSO shadeops	**2**	**2**	**0**

Table 6.4: Shader computation profile for the Short Fur surface shader.

6.2.4 Short Fur Surface

Short fur is a complex surface shader used in production at Industrial Light + Magic to simulate the appearance individual hairs. It simulates in great detail the appearance of each individual hair, including a later of dirt on the tips of the hairs.

Our analysis achieves nearly a sevenfold reduction in program size, it eliminates all loops, more than 80% of all conditionals, and more than 95% of all conditional data-dependence. It eliminates all DSO shadeops and message passing routines other than those to the light source. The naive analysis is unable to eliminate either of the two DSO shadeops, nearly half the conditionals, or the displacement message passing routine. All subsets of the shader contain mod arithmetic which cannot be efficiently mapped to the GPU.

Like the prior surfaces, and like most complex surface shaders, this shader invests substantial computational complexity in the evaluation of procedural texturing and modeling over the surface. However, unlike either parquet plank, the short fur surface computes procedural surface properties specifically for their effects on illumination. Thus, a larger portion of its more complicated constructs depend on the illumination of the surface. This is a relatively extreme example of complex light dependence for a practical shader.

6.2.5 Uberlight

Uberlight is an implementation of a highly flexible and generalized light shader described by Ronen Barzel for feature film-level lighting design. It includes a large and diverse set of

Shader Complexity	Complete	Varying	Dynamic
Instruction Count	**296**	**114**	**100**
Arithmetic	**277**	**103**	**91**
Unhandled	0	0	0
Texture	**2**	**2**	**2**
Conditionals	**17**	**9**	**7**
Data-dependent	17	8	0
Loops	**0**	**0**	**0**
Message Passing	**0**	**0**	**0**
DSO shadeops	**0**	**0**	**0**

Table 6.5: Shader computation profile for the Uberlight light shader.

controls reminiscent of a studio spot light, including a variety of geometric blockers, gels, and spotlight angle parameters. This implementation was provided by Larry Gritz.

Our static/dynamic analysis reduces the static computation profile of the uberlight by a factor of 3. It reduces the number of conditionals by a factor of 2.5, but eliminates all data-dependence of the remaining dynamic conditionals. While our analysis is primarily intended to eliminate the large light-independent computation often contained in surface shaders, the substantial reduction in complexity of this common production-level light shader indicates that substantial additional performance can be gained by specializing the light shader computation.

It is notable that the naive varying analysis can achieve much of the same reduction as our fully informed static/dynamic analysis. This indicates that the shader performs substantial one-time setup which need not be recomputed. Such static initialization computation is where we would expect to achieve any performance improvement for light shaders, for the majority of their parameters are dynamic during lighting design. These results suggest consideration of more aggressive specialization of light shaders against smaller subsets of simultaneously configurable parameters.

6.3 Compiler Performance

Though it is not the central focus of our optimization, our compiler is designed to achieve high performance. We evaluated its performance using the Unix time command on an ancient dual 366mhz Celeron PC running RedHat 9. The test binary was compiled in debug mode in gcc 3.2.2. Under these conditions, our most resource-intensive compile required

2.23s of user time to parse and analyze the warrior shader for the warrior bug test scene. Analysis of smaller shaders required under 1s. The simple standard plastic shader required 0.23s to parse and analyze, while the constant shader required 0.19s, suggesting an overhead of approximately 0.2s to set up and tear down the compiler executable.

The analysis engine achieves high performance through efficient use of the dynamic programming algorithm which implements the reachability analysis. While at least one reachability graph node exists for each node in the shader's abstract syntax tree, the compiler only analyzes the flow of a single value through the program — the virtual *dynamic* value. Thus, reachability queries are only ever made with this node as the source. This allows all other nodes to be more compactly represented in the dynamic programming algorithm by which context-free reachability is computed efficiently, improving the performance and scalability of the analysis.

Such high compilation performance suggests the feasibility of applications involving dynamic re-specialization of shaders or the specialization of new shaders during interactive use. On a more modern machine, implemented in a library which maintained a persistent pre-parsed representation of the shaders, dynamic re-specialization would likely be imperceptible to the user.

Chapter 7

Limitations

In spite of its great initial promise, our technique still faces several limitations in achieving the full generality necessary to support all possible RenderMan features.

7.1 Global Illumination

Global illumination algorithms violate the data flow model of rendering on which the our approach relies. We can largely ignore global interreflection and caustics. Such effects are still rarely used in high-end film production because they are very difficult to control.

However, although not always considered as such, shadows are a global illumination effect. Shadows are the result of an on-demand visibility test *during the end of shading* — long after transformation and camera visibility determination are performed in the primary rendering process. In the data flow model of rendering, this creates an arbitrary loopback to the beginning of the rendering data flow from the end — from shading to the initial global scene representation. This fundamentally violates the principles of a shading-only re-renderer.

We can address shadows in part by proposing our approach as a partial solution to lighting design. The vast majority of lights in most production scenes are not shadow casting, providing only shading. Lighting could still be previewed as described so long as only non-shadow casting lights were being moved. Even shadow casting lights could have all parameters other than light position rendered dynamically, as described. Furthermore, in most scenes, even shadow casting lights could be initially configured and positioned without shadows being enabled. Then the final fraction of the lighting design process could occur

as it does now, using slow offline preview renders. However, the appearance of some scenes — for example, those prominently featuring hairy objects — can be highly dependent on the presence of shadows, making it difficult to light such scenes at all without an accurate preview of shadows cast by the lights. Worst of all, such scenes also tend to be those for which shadow maps are most expensive to recompute. While new shadow maps might be recomputed relatively quickly for many objects of moderate complexity, computing a shadow map for hair or other such detailed microstructures routinely takes minutes.

Shadows represent a fundamental challenge to our approach. However, we believe that a creative combination of fast software ray tracing and progressive refinement will allow effective shadow preview within this framework, while maintaining the advantages of interactive performance for which the system is designed. This is a critical subject of future work.

7.2 Non-linear Lights

Because the computation of shading is specified using arbitrary arithmetic, RenderMan shaders need not necessarily obey the linearity of light observed in the physical world. Non-linear lights cannot be isolated from one another for independent re-rendering as simply as linear lights. Because most reflection functions used in production aim for relative physical realism, real-world shaders commonly conform to the simple linear reflection model.

However, some simple shader operations can violate the simple linearity of the re-rendering shaders. If a single independent term is added into the reflected color after the integration of illumination over the lights, the shading due to each light would no longer be simply linearly independent. A simple emissive term would thus violate the simplistic model of linearly independent light contributions.

In some sense, the common ambient() term used in many shaders thus violates the linearity of a compound illumination expression. It is independent of all non-ambient lights, but the data flow analysis detects its dependence on Cl. In the renderer, the distinction between ambient and directional lights must be handled with special internal logic. The re-renderer must similarly distinguish between ambient lights which contribute only to ambient illumination terms and directional lights which contribute to all other illumination terms.

In the more general case of arbitrary post-integration arithmetic, the compiler could

detect such violations to the simple linear lighting model and factor the re-rendering computation at the corresponding points. Then, the shading due to each inactive light would not be fully computed and summed, but rather the most complete possible illumination terms would be computed for each inactive light, and these terms would be input to the re-rendering shader which would compute the contributions of the current light to the illumination terms and then combine these as necessary to compute the final shading. In cases where multiple independent illumination terms must be combined in a non-linear fashion, this would require either recomputing the shaders for the inactive lights multiple times for each term, or employing multiple output buffers for the different terms, and is a subject of future investigation.

7.3 Transparency & Multi-sampling

Transparency violates the simplicity and regularity of the deep-framebuffer structure by requiring the computation and blending of multiple independent shading samples at a given pixel. Furthermore, transparency creates constraints on the order in which samples must be layered and blended to produce correct results.

In the regular grid structure of the deep-framebuffer, transparent samples must be represented by multiple layers of deep-framebuffer channels which are shaded and blended according to their opacity to produce the final shaded result. Because arbitrary numbers of transparent layers may exist at each deep-framebuffer location, such an extended deep-framebuffer could be of indeterminate size.

Building such a structure would require an extension to the present precomputation mechanism which renders single-layered images of each deep-framebuffer component output from the precomputation shaders. The precomputation shaders could be extended to recursively precompute deep-framebuffer layers wherever the surface opacity Oi was computed to be partially transparent. This condition could initiate a recursive `trace()` along the view direction to precompute shading for the next visible surface, and so forth into the scene. To support storage into a multi-layered deep-framebuffer structure, these shaders would either have to support multiple sets of deep-framebuffer channels — one for each allowable layer of recursion through a transparent surface — or they would have to use an external output mechanism, implemented through DSO shadeops, which stored variable-sized sets of deep-framebuffer layers at each pixel location.

Multisampling effects such as spatial antialiasing and motion blur can at times play a critical role in the appearance of lighting in a scene. Extreme motion blur, for example, common in special effects shots, can radically alter the appearance of sharp highlights and other illumination features which the lighting designer wishes to control. Computing such effects in the deep-framebuffer preview similarly requires multiple deep-framebuffer shading samples per pixel. To represent efficiently, they also require the same irregular number of samples at each location. However, the maximum number of samples per pixel can be more easily bounded for multisampling effects than for transparency.

7.4 Atmospherics

Atmospheric effects such as fog can be critical to the mood and appearance of a scene. In such situations, they are therefore essential to the lighting designer's work, even if they are not configured and modified during lighting design.

Atmospherics are not addressed by our current system, but they could conceivably be integrated into our existing approach by allowing additional attenuation computations between the light and the surface, and between the surface and they viewer. Atmospheric effects are represented in RenderMan using volume shaders, defined using the same shading language constructs we have already analyzed in detail. Atmospheric attenuation between the light and the surface and between the surface and the viewer is computed based on the position of the two points between which light is being attenuated. In the surface/viewer case, these positions are static during lighting design, and so this attenuation could be computed statically and output as a channel in the deep-framebuffer. The position of the light is not static during lighting design, but many atmospheric attenuation effects primarily depend on the distance over which light is being attenuated, and can therefore likely be computed dynamically at moderate cost. Additionally, many parameters to the volume shader will be static with respect to the dynamic light parameters. This static computation could be analyzed using the same techniques already presented. Based on this analysis, volume shaders could be specialized and compiled into the re-rendering process similarly to surface shaders, attenuating Cl between the light and the surface and Ci between the surface and the viewpoint.

Still, in spite of its current limitations, practical study has proven our technique eminently useful in most real-world scenes.

Chapter 8

Conclusion

8.1 Contributions

This thesis has presented a novel solution to the so-called lighting design problem. By exploiting programmable graphics hardware for real-time shading, it achieves far greater interactive performance than any prior technique, accelerating lighting design preview by more than three orders of magnitude over conventional approaches. By employing compiler analysis techniques for automatic shader specialization, and by performing all precomputation directly within the final-frame software renderer, our approach can more consistently provide mathematically accurate results than prior practical techniques, while requiring substantially less maintenance and extension to support future revisions of the final-frame renderer and its interfaces. This solution is above all designed to be practically applicable to actual computer graphics production. Expressly designed around the RenderMan Interface and RenderMan Shading Language used for most high-end production, it is intended to be immediately applicable to real production pipelines, without modification.

8.2 Status

The implementation described in this thesis is academically sound, but practically somewhat incomplete. The implementation has been completed to all points necessary to explore and verify the validity of the presented techniques, but it does not yet represent a full, push-button lighting preview and design pipeline. In particular, the code generation back-ends to the compiler are not yet fully general. Nevertheless, all details of their implementations

have been fully explored at a very low level. The analysis engine and the real-time lighting framework are fully implemented as described, and have been used in producing the results presented, including all shader profiling data.

8.3 Future Work

The major practical applications of this research beg substantial future work to remove any limitations and improve the possibility of its widespread adoption in production. We are beginning to explore several future extensions of this approach to interactive rendering for cinematic lighting design.

Shadows First and foremost, to provide true final-quality feedback, any lighting system must include shadows. Exploration of various possible methods for real-time shadow calculation have suggested that fast ray tracing methods may be the most fruitful avenue for future work. With the minor practical restriction that all shadow map arguments to shadow() calls be physically meaningful — that the shadow maps be rendered from the actual perspective of the light — we can replace all shadow() calls in the re-renderer with ray traced shadow queries.

Ray tracing is highly advantageous in real-time applications because it allows arbitrary selective sampling, to support progressive refinement critical to real-time feedback, because it scales effectively with the irregular, global occlusion queries necessary for shadow computations, and because it does not suffer from the resolution limitations of shadow maps which require far more complicated structures such as deep shadow maps. Software-based real-time ray tracers have reached a stage of development where they can perform millions of ray intersection tests per second against scenes with millions of triangles of static geometry using commodity PCs. Such rates are sufficient to allow shadow tests which resolve to a useful level of detail far faster than the 10+ minutes which may often be required to compute a useful shadow map for a complex scene.

Tight Packing of Deep-Framebuffer Values Because GPU application scalability is most often bound by memory bandwidth and memory size constraints, it seems fruitful to explore methods for reducing the size of deep-framebuffer caches by more tightly packing data into the structure. Nearly all values output from the precomputation pass contain either one or three components, while OpenGL textures store four components per texel.

Thus just tightly packing three-component RenderMan vectors into fewer four-component would yield a 33% reduction in storage and bandwidth consumed by the deep-framebuffer. Further packing one-component values into empty channels in four-component vectors would yield a greater improvement, still. Because all deep-framebuffer channels are accessed regularly, with the same texture coordinates, packing in this way does not cause unrelated data packed into the same texture to be accessed unnecessarily, so all improvements in storage should directly correlate to reductions in bandwidth.

Subsurface Scattering In practice, subsurface scattering represents the most critical single use of DSO shadeops in light-dependent shader computations. Efficient computation of subsurface scattering requires acceleration structures more advanced than can be implemented directly in the RenderMan Shading Language, and therefore must be implemented in external C libraries. By nature, subsurface scattering is fundamentally light-dependent and must therefore be recomputed during re-rendering, unlike many external DSO calls.

Efficient subsurface scattering methods in common usage employ a two-pass rendering mechanism in which many simple irradiance samples are first precomputed over the scattering surface, and then these samples are subsequently integrated using a diffusion kernel over many lambertian shaded surface samples [Jensen and Buhler, 2002]. In a scene with static geometry, the first pass of this approach is fundamentally similar to a deep-framebuffer render in which the sample points are distributed over the scattering surface rather than representing the image plane at the camera's viewpoint. A similar two-pass render could then be employed in the deep-framebuffer architecture whereby the irradiance samples are computed with a fast lambertian shading of the surface-space deep-framebuffer, and these samples are diffused at each image-space deep-framebuffer location to quickly compute an accurate subsurface scattering term. While external C libraries cannot generally be mapped to the GPU, subsurface scattering is commonly implemented via a single, fixed library used across many productions at a given studio, so it can feasibly be special-cased and recognized by the compiler, and replaced with such an approximation for re-rendering purposes.

User Interface Implications The interactive performance afforded by our technique represents a radical transformation of the lighting design process. It will be most interesting to study the impact that this revolution in performance and interactivity has on the practical workflow of real-world lighting designers. It will likely be useful to devise new

tools, interfaces, and features which exploit interactive performance.

Some interface features of interest include tools for controlling multiple related lights in tandem and for the automatic creation of diffuse bounce lights which often make up many of the lights in a scene. Techniques such as instant radiosity [Keller, 1997] could likely be employed to leverage the high interactive performance of the re-renderer with non-shadow casting lights, while creating very complex illumination effects which can still be previewed interactively and controlled by the lighting designer as individual bounce lights.

Additionally, it may be useful to allow additional, non-lighting parameters to be edited by the lighting designer. Parameters such as surface roughness, which determine specular highlight size, are intimately related to the appearance of lighting in a scene, and so could therefore be useful for the lighting designer to edit. Our compiler framework is sufficiently general to support an arbitrary selection of user-configurable parameters, so long as the resulting re-rendering shader can be mapped to the GPU. However, the interactive rendering framework currently only exposes one set of editable parameters, for the currently active light, and would have to be extended to support separate configurable parameters for each surface.

Ultimately, our foremost goal is to facilitate the greatest possible adoption of our techniques. The fundamental shift from slow, offline rendering to smooth, interactive feedback stands to transform the creative process of lighting design. By enabling artists to directly interact with their scenes at full, final quality, we hope to transform both the quality and economics of computer graphics production. Thus, above all, we wish to improve and promote our work in production to maximize the real-world use of our transformative techniques for lighting design.

Appendix A

The RenderMan Shading Language

The shading language is a special-purpose, C-like programming language, orthogonal to the scene description interface and specially designed for the computation of shading and lighting. It includes support for most standard C-style expressions and control flow constructs, including arithmetic and logical operators, as well as loop and conditional constructs. It supports the logical interpretation of most expressions for all built-in numeric types, including vector and matrix values. It additionally includes basic arithmetic operators for vector dot- and cross-products. It includes a standard library of functions supporting a wide variety of predominantly mathematical operations useful for graphics.

A.1 Types

The shading language defines a small selection of built-in types relevant to graphics computation. All types are opaque, abstract data types, and pointer types are not supported.

float All numerical computation is performed on floating point values. The shading language contains no integral types. Thus, floats are the only supported scalar type. Their precision is not specified and is assumed to be sufficient for high quality rendering.

point The original shading language specification defined a single point type which was used for all 3-component geometric vectors. Later revisions of the language added two additional classes of geometric 3-vectors: vector and normal. All types are represented as triples of float values, but the more detailed typing allows subtly different semantics to be

applied to each geometric type, specifically during transformation.

color The shading language represents color values as an opaque type. Though colors are commonly represented as RGB triples, colors may contain arbitrary numbers of components, depending on the requirements of the color space in which a given color is represented.

matrix The shading language includes a matrix type representing a 4x4 homogeneous transformation matrix which may be used to transform vector values between coordinate spaces.

string The shading language supports strings as an opaque type. Strings have no corresponding array or pointer interpretation, and characters do not exist as a distinct subtype. According to the original language specification, strings were only used for naming of external entities, such as parameters, texture maps, or coordinate spaces. In this original specification, strings were required to either be constants or shader parameters — never locals. More recent extensions of the shading language have added support for string construction and manipulation, creating dynamic string types, but their use remains primarily for the naming of external entities.

Arrays The shading language supports a subset of C-style array functionality for collections of arbitrary types. Arrays are opaque objects — they have no pointer interpretation. All arrays must be of compile-time static dimension, except for arrays passed as parameters to a shader. Multidimensional arrays are not supported.

A.2 Shaders

The purpose of programming in the RenderMan Shading Language is the definition of shaders. Shaders are function-like constructs which implement one of five interfaces — surface, light, displacement, volume, or imager — defining how and where they are bound during the rendering process. Taken together, these interfaces define data flow modules which combine to make up the shading process of the rendering data flow.

The data flow is defined along specific data values input to and output from each module in the process. These data flow values are represented in the shading language as implicitly-defined global variables. Each shader interface allows read-only access to a variety of relevant values at the current shading sample, and exposes write access to the value

or values which shaders implementing that interface are to compute. All data flow inputs to each shader are computed and supplied by the renderer during rendering. The parameters to a given instance of a shader, however, are represented as function-style arguments. These parameters are specified and bound to the given shader when it is instantiated and applied to a given object during scene description.

The interfaces defined for each class of shader clearly define and abstract the interaction between shaders at different stages of the data flow, allowing arbitrary combinations of shaders to be used together.

For simplicity, we will only consider shaders directly related to the light-dependent portion of the shading data flow: surface and light shaders.

surface The role of the surface shader is to compute the color and opacity of each point on a surface as perceived from the perspective of the viewer. The surface interface provides read access to the position, normal and all other standard parameters over the surface, interpolated at the current sample. Surface shaders provide a local mapping from these interpolants to the visible color at that sample.

Because the visible color of a surface depends on its illumination by all light sources casting light onto it, the surface interface allows the use of an additional construct known as illuminance by which a shader can integrate the contribution of each light casting onto it. The illuminance construct is defined like a specialized loop. Whereas the bounds of a conventional C-style loop are expressed with a conditional expression, illuminance bounds its integration by a cone — geometric bounds within which all light sources will be integrated. The body statements of the illuminance loop compute the illumination reflected from the surface due to each incident light. Within the loop, the additional data flow variables L and Cl, representing the direction and color cast on the point by the light over which the integration is currently iterating. This construct thus connects the data flow from light to surface shaders.

light The role of the light shader is to compute the contribution of the corresponding light to each point illuminated by that light. For our purposes, we will consider light shaders to be implicitly invoked to compute their contribution to the light incident on the point currently being shaded by a surface shader.

Just as the surface interface facilitates integration over the contribution of lights using

the illuminance construct, so too the light interface includes two specialized loop constructs which facilitate the connection of data flow between lights and surfaces. illuminate and solar both similarly specify loops over points being illuminated, as bounded by a cone of illumination. Within the body of these loops, the position P of the point currently being shaded is defined, as is the writable value Cl, representing the color and intensity of light contributed to that point as computed by the light shader. illuminate and solar operate similarly, except illuminate defines a local light source, while solar defines a distant, purely-directional source. Values assigned to Cl outside of these per-point loops are treated as ambient illumination.

A.3 Functions

The shading language supports C-like function definitions. Unlike in C, all arguments are passed by reference, not by value. As a further simplification of C function definition, all non-void functions must include exactly one return statement.

Many practical RenderMan implementations — in particular, Pixar's PRman and the popular freeware BMRT renderer — inline all function calls during shader compilation. Thus, while the language does not explicitly disallow recursive function definition, the limitations of practical implementations mean that few, if any, shaders are written using recursive functions.

The language defines a specialized class of built-in functions for message-passing between the shaders bound together to shade a single sample, and between these shaders and the host renderer within which they are executing. These functions conditionally assign the value of a parameter or attribute named by a string argument into an object argument which is passed by reference, predicated on the existence of a like-named parameter or attribute sharing the same type as the destination object in the context into which the given message passing call is made.

Bibliography

[Andersen, 1996] Peter Holst Andersen. Partial evaluation applied to ray tracing. In W. Mackens and S.M. Rump, editors, *Software Engineering in Scientific Computing*, pages 78–85. Vieweg, 1996.

[Apodaca and Gritz, 2000] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: creating CGI for motion pictures*. Morgan Kaufmann, 2000.

[Bleiweiss and Preetham, 2003] Avi Bleiweiss and Arcot J. Preetham. Ashli – Advanced shading language interface. In *Proceedings of the 30th annual conference on Computer graphics and interactive techniques*. ACM Press, 2003.

[Cutts et al., 1999] Q.I. Cutts, R.C.H. Connor, and R. Morrison. The PamCase Machine. In *Fully integrated data environments*. Springer, 1999.

[Dietrich, 2004] D. Slim Dietrich. Shader Model 3.0 – No limits. Technical report, NVIDIA, 2004.

[Gershbein and Hanrahan, 2000] Reid Gershbein and Pat Hanrahan. A fast relighting engine for interactive cinematic lighting design. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 353–358. ACM Press/Addison-Wesley Publishing Co., 2000.

[Guenter et al., 1995] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 343–350. ACM Press, 1995.

[Hanrahan and Lawson, 1990] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298. ACM Press, 1990.

[Hanrahan, 1983] Pat Hanrahan. Ray Tracing Algebraic Surfaces. In *Proceedings of SIGGRAPH 83*, 1983.

[Hart, 2001] John C. Hart. Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM Press, 2001.

[Jensen and Buhler, 2002] Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. In *Proceedings of the 29th annual ACM symposium on principles of programming languages*. ACM Press, 2002.

[Jones et al., 1993] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[Keller, 1997] Alexander Keller. Instant Radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 1997.

[Knoblock and Ruf, 1996] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, 1996.

[Mark et al., 2003] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics

hardware in a C-like language. In *Proceedings of the 30th annual conference on Computer graphics and interactive techniques*. ACM Press, 2003.

[Mogensen, 1986] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Perlin, 2004] Ken Perlin. *GPU gems: Programming techniques, tips and tricks for real-time graphics*, chapter Implementing improved Perlin noise. Addison-Wesley, 2004.

[Pixar, 2001] Pixar. Irma. `https://renderman.pixar.com/products/tools/irma.html`, 2001.

[Reps et al., 1995] Thomas Reps, Susan Horowitz, and Mooly Sagiv. Precise interprocedural data flow analysis via graph reachability. In *Proceedings of the 22nd annual ACM symposium on principles of programming languages*, pages 49–61. ACM Press, 1995.

[Upstill, 1989] Steve Upstill. *The RenderMan companion: a programmer's guide to realistic computer graphics*. Addison Wesley, 1989.