

The Lightspeed Automatic Interactive Lighting Preview System

by

Jonathan Millard Ragan-Kelley

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

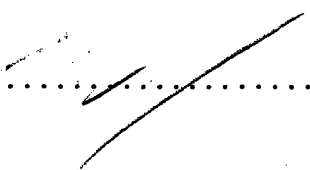
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

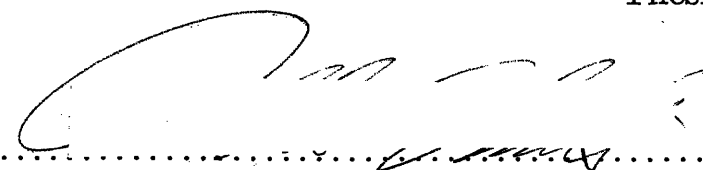
May 2007

[June 2007]

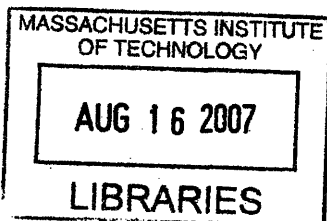
© Massachusetts Institute of Technology 2007. All rights reserved.

Author 
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by 
Frédo Durand
Associate Professor
Thesis Supervisor

Accepted by 
Arthur C. Smith
Chairman, Department Committee on Graduate Students

ARCHIVES



The Lightspeed Automatic Interactive Lighting Preview System

by

Jonathan Millard Ragan-Kelley

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

We present an automated approach for high-quality preview of feature-film rendering during lighting design. Similar to previous work, we use a deep-framebuffer shaded on the GPU to achieve interactive performance. Our first contribution is to generate the deep-framebuffer and corresponding shaders automatically through data-flow analysis and compilation of the original scene. Cache compression reduces automatically-generated deep-framebuffers to reasonable size for complex production scenes and shaders. We also propose a new structure, the *indirect framebuffer*, that decouples shading samples from final pixels and allows a deep-framebuffer to handle antialiasing, motion blur and transparency efficiently. Progressive refinement enables fast feedback at coarser resolution. We demonstrate our approach in real-world production.

Thesis Supervisor: Frédo Durand
Title: Associate Professor

Acknowledgments

This thesis is derived from a project done in collaboration with Charlie Kilpatrick, Brian Smith, Doug Epps, Paul Green, Christophe Héry, and Frédo Durand. The final implementation was a joint effort with Charlie, Brian, and Paul, in which they contributed major ideas central to the successful application in production. Charlie and Brian are responsible for cache compression, and both centrally contributed to the scalability of the system including tiling, texture atlases, and the interaction of light caching with progressive refinement. Charlie was instrumental in bringing the compiler approach to production quality, addressing cache-required code, exceptions, dynamic loop unrolling, message passing, and other issues. Doug also wrote code, and he, Christophe, and Frédo were centrally involved in many stages of design. Frédo has been intimately involved in the crafting of the paper. Other graduate students should be so lucky.

Numerous additional people have contributed to this project in its many years of exploration and implementation.

This work started under the advising of Pat Hanrahan, initially in collaboration with Ujval Kapasi. Alex Aiken and John Kodumal proposed dependence analysis by graph reachability and provided the first analysis library we used. Matt Pharr, John Owens, Aaron Lefohn, Eric Chan, and many members of the Stanford and MIT Graphics Labs provided years of essential advice and feedback.

Tippett Studio took great risk in actively supporting early research. Dan Goldman introduced the work to ILM, where Alan Trombla, Ed Hanway, and Steve Sullivan have overseen it. Many developers have contributed code, including Sebastian Fernandez, Peter Murphy, Simon Premože, and Aaron Luk. Hilmar Koch, Paul Churchill, Tom Martinek, and Charles Rose provided a critical artist's perspective early in design. Dan Wexler, Larry Gritz, and Reid Gershbein provided useful explanations of commercial lighting technologies.

We thank Michael Bay for graciously sharing unreleased images from his movie, Dan Piponi for generating our hair data, and the anonymous reviewers for their

insightful discussion and criticism. Sylvain Paris, Ravi Ramamoorthi, Kevin Egan, Aner Ben-Artzi, and Kayvon Fatahalian provided critical writing feedback.

This work was supported by NSF CAREER award 0447561, an NSF Graduate Research Fellowship, NVIDIA Graduate Fellowship, Ford Foundation Graduate Fellowship, Microsoft Research New Faculty Fellowship and a Sloan fellowship.

Chapter 1

Introduction

Configuring lights is a critical bottleneck in modern production rendering. Recent advances have sought to provide real-time preview using deep-framebuffers and graphics hardware [7, 23]. A deep-framebuffer caches static values such as normals and texture samples in image space, and each time the user updates light parameters, real-time shaders interactively recompute the image from the cache. Unfortunately, such approaches require substantial additional work from shader authors. For example, in the *lpics* system deployed at Pixar [23], at least two versions of each shader need to be written in place of just one: the usual RenderMan shader used for the final rendering (with additional code paths to cache data), and a Cg version used for real-time preview.

Automatic shader specialization has been proposed for simplified shading languages with no control flow [8], but, in the practical context of production, Pellacini et al. [23] concluded that, while they “believe that automatic translation might be a superior solution in the future, current approaches are not ready for use.”

This is the first challenge that our work tackles: we alleviate the need to author multiple versions of a shader by automatically translating unmodified production RenderMan shaders into real-time shaders and precomputation shaders. This translation is part of a larger process that automatically generates deep-framebuffer data from unmodified existing scenes. In theory, some RenderMan code cannot be translated into GPU shaders, but we have found that, in practice, the dynamic parts of our production shaders translate well.

In contrast to pure static compiler analysis, we use post-execution cache compression to supplement a relatively simple compiler analysis. Cache compression effectively reduces automatically-generated deep-framebuffers to reasonable size for complex production shaders.

In addition, transparency, motion blur and antialiasing can be critical to judge appearance. We introduce the *indirect framebuffer*, which enables these effects without linearly scaling rendering time. Similar to RenderMan, it decouples shading from visibility, but also precomputes the final weight of each shading sample for the relevant final pixels. Given the complexity of shots that we handle, we also use progressive refinement to offer both interactive feedback (multiple frames per second) and faithful final quality (potentially after a few seconds).

Finally, it is important to facilitate the implementation of new passes in a preview system. We use a computation graph that directly expresses the dependencies and data-flow between passes to implement shadows and translucency, while maintaining orthogonal extensibility for future multipass effects.

Contributions We make the following contributions:

- We introduce **compiler techniques** to automatically translate an original, unmodified RenderMan shader into a precomputation shader that caches values in a deep-framebuffer, and a shader for real-time preview.
- We introduce the **indirect framebuffer** to efficiently support transparency and multisampling (including antialiasing and motion blur).
- We show that **cache compression** through static preprocessing of deep-framebuffer caches is an effective approach for managing the visibility and shading complexity of antialiased, motion-blurred, and transparent scenes.
- We use a **computation graph** architecture to encode the dependencies between different computations during multipass preview rendering with shadows, subsurface scattering, and progressive refinement.
- We describe a full **production relighting system** that is being deployed in two studios with different rendering workflows. We discuss major challenges and

design goals, in particular real-time performance, seamless integration in existing pipelines, and ease of implementation and maintenance.

1.1 Prior Work

Fast relighting has long been a major area of research [6, 18]. Software renderers can be optimized for repetitive re-rendering by caching intermediate results at various stages of the rendering process, as pioneered by TDI in the 1980s [1, 24, 19, 29]. However, such optimizations must be integrated at the core of a rendering system and are still far from interactive for film scenes.

Séquin and Smyrl [27] introduced a parameterized version of ray tracing that enables the modification of some material and light properties after precomputation (although not the light direction or position). They also perform cache compression.

Gershbein and Hanrahan created a system for lighting design [7] which cached intermediate results in a deep-framebuffer inspired by G-Buffers [26]. They cached a fixed set of data, and approximated shading with multitexturing. Pellacini et al. performed shading on programmable graphics hardware [23] using manually-written shaders that emulate RenderMan shaders. These systems require manual segmentation of shaders into light-dependent and light-independent components, and manual translation of preview shaders. While this allows for manual optimization to maximize preview performance, it is a significant burden. We chose to potentially sacrifice performance but tremendously improve pipeline integration and maintainability by automating the segmentation and translation of shaders. Furthermore, we extend prior deep-framebuffer systems by enabling the efficient rendering of transparent surfaces and multisampling effects, such as motion blur. Finally, our approach also automatically supports editing many (user-selected) surface properties because it employs data-flow analysis with respect to arbitrary parameters.

Wexler, et al. implemented high-quality supersampling on the GPU [30], but they focus on final rendering, while we optimize for static visibility, resulting in a different data structure. We build on recent work on direct-to-indirect transfer, which exploits linearity for global illumination in cinematic relighting [11]. We apply similar

principles to multisampling, transparency and subsurface scattering.

Jones et al. segmented shaders into static and dynamic subsets and cached shading information in texture-space to accelerate rendering the same scene multiple times under similar configurations [14]. However, their technique only cached shading computation—not tessellation, displacement, etc.—and required manual shader segmentation.

Our goals cannot be met fully by pre-computed radiance transfer (PRT) techniques [28, 18], because they usually make assumptions on the reflectance or lighting and have significant precomputation cost (many times the cost of a single final-frame render). In contrast, we need to handle the effect of local point light sources and arbitrary reflectance, and cannot afford such expensive precomputation. Furthermore, computing illumination, itself, is a large part of our run-time calculation, as production light shaders are very complex.

Compiler specialization of graphics computation was first used for ray tracing [9, 17, 2]. Guenter, Knoblock & Ruf developed data specialization to reduce the cost of recomputation when only certain shading parameters vary, by automatically segmenting shaders into parameter-dependent and -independent components [8, 15]. We leverage their approach in the context of lighting design and extend their analyses to global data-flow through existing real-world RenderMan shaders. We solve specialization using a graph formulation, mentioned but not implemented by Knoblock and Ruf [15]. This allows us to not only specialize with respect to dynamic parameters, but also to perform dead-code elimination and other analyses, all from a single dependence analysis.

Olano and Lastra mapped RenderMan-like shading onto an architecture akin to a modern programmable GPU [20]. Peercy et al. [21] and Bleiweiss and Preetham [4] addressed the compilation of RenderMan shaders onto graphics hardware. We, too, exploit the fact that a large subset of the RenderMan Shading Language (RSL) can be compiled to a GPU. Our interest, however, is not in using RSL as a GPU shading language, but in automatically specializing final-frame shaders and creating an appropriate deep-framebuffer for interactive relighting.

Chapter 2

System Design

2.1 Design Goals

Our primary objective is, given a fixed scene geometry, material and viewpoint, to enable the interactive manipulation of all light source parameters, including intensity, position, and falloff, as well as to create and remove light sources. The restriction to lights came first from current production workflow where light source placement is a separate step at the end of the pipeline, after all other aspects have been frozen. We were also motivated by technical limitations: surface shaders tend to be more complex and could prove harder to map to graphics hardware.

However, it also became apparent that our approach can enable the modification of many (but not all) material appearance parameters. We have sought to facilitate this, although only as a secondary objective.

In order to receive widespread adoption in production, a lighting design system must meet the following three major design goals.

High-performance preview Minimizing feedback time is our primary goal. Specifically, we wish to provide:

- **Low-latency feedback** – When the user modifies a light parameter, image refresh must be instantaneous. Final quality might take a few seconds through progressive refinement, but low-latency feedback is critical to seamless user interaction.

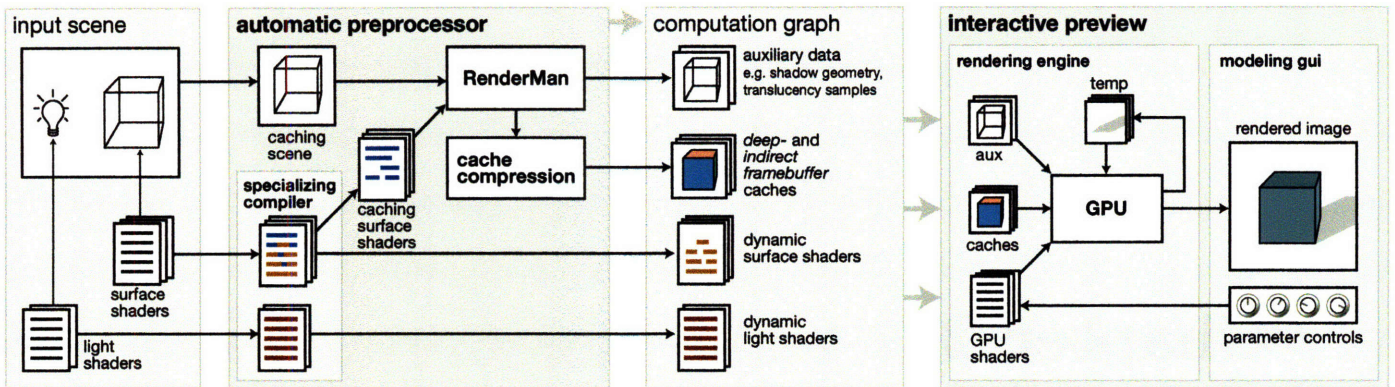


Figure 2-1: *Our system takes as input the original RenderMan scene with its shaders. Our specializing compiler automatically separates a shader into static and dynamic parts and uses RenderMan to cache static computation and auxiliary data. The dynamic part is translated into Cg. Cache compression greatly reduces the size of the cached data. The preprocess generates a computation graph that encapsulates the computation and data binding necessary to re-render the scene. The real-time rendering engine executes the graph to generate intermediate data (shadow maps, etc.) and run the dynamic shaders over the cache on the GPU. The indirect framebuffer enables antialiasing and transparency. The GUI application modifies light parameters through the graph API.*

- **Fast initial precomputation** – To be accepted by artists, this tool should not increase the time it takes to begin work on a shot. We seek to keep the initial preprocessing time as short as rendering one frame with the offline renderer.
- **High absolute rendering speed** – Although secondary to latency and startup time, absolute rendering speed must be optimized.

Seamless integration with existing pipelines A preview system should be transparent to the user and require no additional work to use within an existing pipeline. This means that it should stand in for the existing offline rendering pipeline by:

- Taking the **same input** – unmodified RenderMan scenes and shaders.
- Producing the **same output** – using shading and visibility computation with extremely high fidelity to the final rendering, including antialiasing, motion blur, and transparency.
- Using the **same workflow** – in particular the existing light editing GUI, which

varies from studio to studio. This requires our system to integrate with different GUI software.

Ease of implementation and maintenance Production rendering pipelines are complex and continually evolving. A preview system cannot afford the same implementation investment and should not require major re-implementation whenever the final-frame renderer is updated, the shaders changed, or the pipeline altered. Our system must achieve effective:

- **Reuse** – Our system seeks to reuse the existing pipeline wherever possible, offloading most precomputation directly to the existing offline pipeline.
- **Flexibility** – Our system is developed for two independent studios, with different pipelines and toolsets, so we wish to reuse as much as possible between these two environments.
- **Extensibility** – It should be as easy as possible to support new functionality—from using new shaders to implementing new multipass effects—in a simple, modular fashion.

2.2 System Architecture

Our approach (Fig. 2-1) can be decomposed into an automatic preprocess and a runtime phase that communicate through a dynamically-generated computation graph. We take as input the same RenderMan scene and shaders used for final rendering.

Automatic specialization First, we automatically slice all surface shaders into a static component that can be cached and a dynamic component that will be executed by the real-time engine (Section 3). For surface shaders, we then generate two new shaders: a static precomputation shader, which is executed once in the final-frame renderer to generate a deep-framebuffer cache, and a dynamic re-rendering shader (in Cg), which is executed repeatedly over the deep-framebuffer to generate interactive previews. We directly translate light shaders to execute together with the re-rendering

surface shaders on the GPU.

The automatic specialization of shaders can yield a performance penalty for the interactive preview compared to manually optimized and simplified code [7, 23], but in our context, seamless integration took precedence over final performance. Another potential limitation of automatic translation is that not all RenderMan code can be mapped to the GPU. However, for our production shaders this has been a surmountable challenge.

Indirect framebuffer Our core real-time rendering technique is similar to traditional deep-framebuffer approaches and uses Cg shaders to perform computation on all deep-framebuffer samples on the GPU. However, we introduce a new level of indirection through the *indirect framebuffer* to decouple shading samples from final pixel values, thereby efficiently handling antialiasing, motion blur, and transparency. This indirectly also enables flexible progressive refinement (Chapters 4, 5).

Cache & visibility compression We rely on *static preprocessing* of the cached data to compensate for overestimates of the compiler analysis, as well as to cull the deep-framebuffer and indirect framebuffer based on visibility. This provides at least an order of magnitude reduction in total cached data sizes while allowing the compiler to remain relatively simple.

Multipass rendering We enable multipass effects such as shadow mapping and subsurface scattering. This requires the preprocessor to output additional, auxiliary data, such as geometry needed for shadow mapping or lighting samples for translucency. Although translucency currently incurs substantial cost for our preview, it demonstrates the generality of our architecture.

Computation graph The overall re-rendering algorithm is encoded as a computation graph, generated during preprocessing from the original scene and shaders. The graph provides a specification of how to re-shade an image from the cache under new lighting configurations (Section 6). The computation graph provides two critical

abstractions. First, it encodes dependencies between different elements computed during real-time rendering, which is particularly critical for progressive refinement and multipass effects. Second, the graph abstracts the preprocessing from the editing GUI. So long as the generated graph conforms to certain basic conventions, the preprocessing stage can be updated and extended without affecting the GUI tool. This is important to our design goal of integrating seamlessly with multiple different workflows.

Chapter 3

Automatic Deep-Framebuffer Caching

We wish to automatically generate a deep-framebuffer and real-time preview from an unmodified RenderMan scene. We first determine which parts of the computation are static vs. dynamic with respect to the light parameters. We then create new RenderMan Shading Language (RSL) shaders that compute and output the static values, and use RenderMan to create a deep-framebuffer cache. We preprocess the cache output by RenderMan to compress redundant and irrelevant values. Finally, we translate the dynamic part of the computation into real-time GPU shaders that access the deep framebuffer as textures. Previous work has achieved these steps manually. Our contribution is to make this process fully automatic.

3.1 Data-flow Analysis for Specialization

We build on techniques from data-flow analysis to label the static and dynamic parts of a shader [12, 25]. We need to conservatively identify all expressions that depend directly or indirectly on dynamic input parameters. This can naturally be turned into a graph reachability problem: an expression in a shader is dynamic if it is “reachable” from a dynamic parameter. RenderMan separates surface and light shaders and we focus on specializing surface shaders, since light shaders are mostly dynamic with

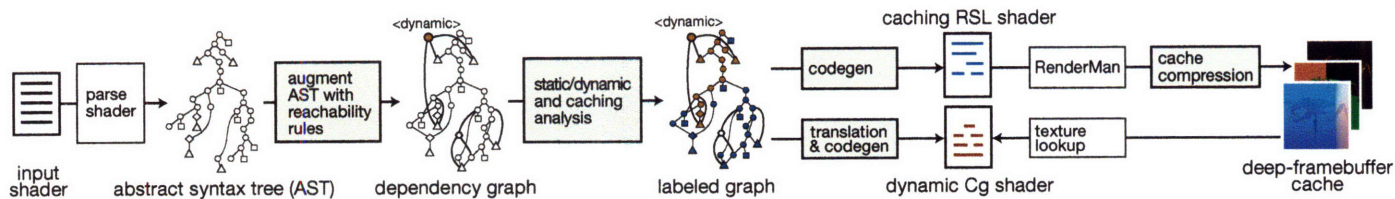


Figure 3-1: Specializing compiler. *The input shader is represented as an abstract syntax tree (AST). We augment it to encode dependence between variables and expressions. We create a special <dynamic> node and connect it to all dynamic inputs. To decide if an expression is dynamic, we query whether it depends on any dynamic parameters by testing if it is reachable from the <dynamic> node. Once the shader has been split, we generate two new shaders, a caching shader (in RSL) and a real-time shader (in Cg). RenderMan executes the caching shader over the scene and the cached values are compressed to generate a dense deep-framebuffer, which is read by the dynamic shader during preview.*

respect to light parameters, and lights have no natural domain for pre-sampling and caching values. We instead directly translate light shaders, and bind them to surfaces at runtime.

3.1.1 Dependence Analysis

To facilitate the analysis of a shader, we parse it into an abstract syntax tree (AST), the standard internal program representation in a compiler. The AST only encapsulates local dependencies within expressions, and we need to add global dependencies introduced by the assignment of variables. The first step of our analysis (Fig. 3-1) adds global dependencies to transform an abstract syntax tree (AST) representation of the shader into a *dependency graph* that encodes all dependencies between expressions. (Note that this is different from our computation graph introduced in Chapter 6).

The new directed graph is a strict superset of the AST and includes additional edges representing dependencies due to assignments, control structures and function calls. We augment the AST by applying a number of simple rules that create appropriate edges for variable assignment and function calls, taking into account control structures such as conditionals and loops.

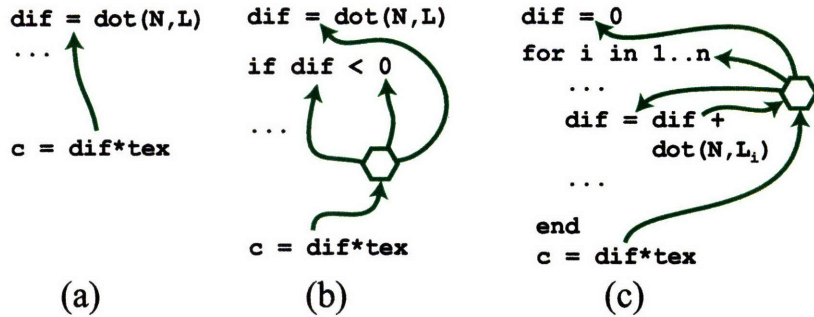


Figure 3-2: (a) The value of a variable depends on its last prior assignment. However, this notion of “last prior” is affected by conditionals (b) and loops (c).

Variables Variable assignments introduce dependences: an expression that uses a variable depends on the last prior assignment to this variable (Fig. 3-2(a)). Ignoring control structures, we simply add an edge connecting each use of a variable to its last prior assignment.

Pointers (and the resultant variable aliasing) create the most challenging problems in traditional data-flow analysis, but the original RenderMan specification [10] does not include pointers, simplifying variable analysis. Pass-by-reference could generate aliasing if a function of multiple variables is called with the same variable used multiple times as an argument, e.g. $f(x, x)$. We have not encountered such cases in practice and can generate an error if we do.

Control-flow Conditionals and loops affect this notion of “last prior” (Fig. 3-2(b) and (c)). They cause branching in the execution, and each branch can produce a different last prior assignment. At every control structure we therefore create new nodes in our graph that act as virtual last prior assignments and depend on the multiple possible prior assignments, as well as the branch conditions.

Loops The iterative nature of loops makes assignment dependencies wrap around from the end of the loop back to the beginning: an assignment at the end of an iteration has influence at the start of the next (Fig. 3-2(c)). In addition, the iteration bounds can influence the final value of any assignment inside the loop. For a variable reference inside a loop, we therefore look for the last prior assignment. If it is also

inside the loop, we create one edge between assignment and reference. If the last prior assignment is outside the loop, we create the corresponding edge, but we also search backwards starting from the end of the loop for any last prior assignment from a prior iteration.

Loop control statements such as `break` and `continue` must be handled too, and additional dependencies are added at the end of the loop to assignments defined before them. RenderMan includes additional constructs—`illuminate`, `illuminate` and `solar`—that integrate over illumination by iterating over all light sources from each surface, and all surfaces from each light. They are treated similarly: their bounds are defined geometrically, but they are logically similar to traditional loops.

Function Calls In RenderMan, function calls are inlined, which simplifies our analysis. For user-defined functions, we simply perform the inlining before analysis, and the dependence graph is constructed without any notion of function calls. Since built-in functions are not represented in the source code, the analysis engine uses a priori knowledge of their implementation to directly insert the appropriate graph at each call site.

The data-flow model of the RenderMan shader interface allows shaders of a given type to access parameters of other shader types using a function-call interface called “message passing.” Our analysis treats all *message passing* calls as static, except for those receiving dynamic arguments or those accessing dynamic values (mostly `lightsource()` calls).

3.1.2 Static/Dynamic and Caching Analysis

Given a graph of dependencies, we add a single new node called *dynamic*. To define which parameters can be edited, we connect it to all dynamic input variables (specified to the compiler by regular expressions matched against parameter names). An expression is static with respect to those parameters only if it is not reached by the *dynamic* node.

Additional rules are applied for *caching analysis* to ensure that the generated

shaders are valid, as described in Knoblock & Ruf’s work [15]. They further propose a range of more complex constraints and code transformations to improve the quality of generated caches by detecting redundant values and trading increased dynamic code to reduce cache size. In contrast, we cannot afford to increase dynamic code, since our shaders are already at the limits of our deployed hardware, and we find that simply post-processing the actual cached data is much simpler and performs better in practice than inherently pessimistic compiler analysis to detect redundancy, since it can detect values which are only redundant due to their dynamic evaluation. (Sec. 3.4).

We perform dead-code elimination using the same dependence graph by connecting output values to a new *output* node. Any expression which does not reach *output* is dead. Dead code exists because shaders often output secondary values during final rendering for use in compositing or other subsequent operations, but which are not necessary during preview.

3.1.3 Cache-Required Code

Our caching analysis constrains dynamic shaders to operations that can be executed on the GPU. We can force certain operations—namely calls to external C routines, and unimplemented shadeops (e.g., `trace`)—to be labeled *cached* even if the dependence analysis labeled them *dynamic*. Static/dynamic analysis eliminates most such operations in our shaders.

We can recognize light-dependent *cache-required* nodes as errors, but we find simply warning the user and computing the values statically at cache time often provides usable preview results. For example, Figure 7-2 used ray traced, as well as environment mapped reflections. While the environment map dominates the appearance, and is easily computed on the GPU, ray tracing is essential for some self reflection effects. We were surprised at first to find that cache-time static reflection values still provided a good preview, since it was the rough *shapes* of the reflected geometry, more than their exact shaded colors, which mattered most to the appearance. In cases where useful preview requires dynamic evaluation of these terms,

we implement dynamic equivalents using multipass algorithms, as we have already done for shadows and subsurface scattering, described in Chapter 6.

3.2 Code Generation and Translation

Once we have decided which computations to cache, and which to execute dynamically during preview, we generate two new surface shaders, one for each phase.

3.2.1 RenderMan Precomputation

Caching computations are emitted as a new RSL shader. Each node labeled *cached* is wrapped in an expression which caches and returns the value of the node, making caching transparent to the rest of the code. The caching shader also outputs bookkeeping data—namely, a unique shader ID and micropolygon ID for each sample.

When branch conditions are dynamic, control flow in the dynamic preview shader may differ from the caching execution. If values are cached inside a dynamic conditional, the caching shader must execute both potential branches. However, this can introduce exceptions (e.g. divide-by-zero) when static branches, used to prevent illegal values, are disabled for caching. This is a major problem in practice which Knoblock & Ruf were unable to solve [15]. We avoid this by recording and restoring the appropriate assignment state before, between, and after the flattened conditional in the caching shader.

We must lay out the data so that caching and dynamic shaders can reliably use a common indexing scheme to access data. We represent cache values at a given deep-framebuffer sample using compile-time static offsets, rather than dynamically incremented indices, to avoid order of evaluation discrepancies. Static indices are problematic when values are cached inside of dynamic loops. In practice, we find such loops in our shaders can usually be given a reasonable static upper bound (e.g. loops over the body of a shader to produce multi-layered materials never use more than a small constant number of layers), which we hint using `#pragmas` to statically unroll them.

Finally, we generate a new RenderMan scene that replaces each shader by its caching equivalent. We run it through RenderMan to generate the deep-framebuffer (Fig. 3-1). Caches are output as pointcloud (ptc) files during baking. Deep-framebuffer values are output per-micropolygon, along with bookkeeping data to uniquely identify each micropolygon and the surface shader bound to each point. Micropolygon IDs are correlated with a separate pixel samples pointcloud to reconstruct the visible point list and build the indirect framebuffer (see Ch. 4).

3.2.2 Cg code generation

Dynamic surface shaders are emitted as new Cg shaders which read the deep-framebuffer cache as textures.

The key issue in translating RSL to Cg is to mimic RenderMan's richer data-flow and execution semantics. Communication of light color and direction between surfaces and lights is accomplished through shared global variables, as in RSL. However, RSL also allows surfaces and lights to access each other's parameters by name through *message-passing*. We implement this by communicating parameters through global variables.

We represent string tokens, including message passing identifiers, by encoding static string values in floats using unique IDs. This enables runtime code to pass and compare (though not modify) strings on the GPU. RSL also uses strings to represent transforms and texture handles, so our Cg string type includes the necessary texture samplers and matrices for all major uses of strings.

Finally, RSL supports the computation of arbitrary derivatives over the surface. Cg also supports derivatives, but its fast approximations are low-quality. In practice, we find that high quality derivatives are only significant in dynamic code when used for large texture filter kernels. These primarily depend on surface partial derivatives, which are not dynamic, so we simply cache them when necessary.

3.2.3 Light translation

While surface shaders are specialized, light shaders are directly translated through the same Cg code generator. Similar to RenderMan, we generate Cg light and surface shaders separately and combine them at load time. They communicate primarily through Cg *interfaces* [16].

This approach can only automatically translate light shaders which do not rely on *cache-required* functionality—most significantly, external C calls. (Ray tracing may be used in lights to compute shadows, but we can replace these with shadow maps during preview.) In practice, our lights only call C DSOs for simple operations like fast math routines, which are trivially replaced with native instructions on the GPU, so we have not found this problematic.

3.3 Specialization Results

Figure 3-3 summarizes the results of our shader specialization approach. Note that the dynamic shader complexity depends on both the light and surface shaders. Generic Surface is a multipurpose “übershader” that forms the basis of most of our custom shaders. However, it does not result in dramatically larger dynamic shaders than a simpler surface because most of the code is static and dynamic code is dominated by lighting computation. For comparison, note that RSL instructions tend to be higher-level, and the equivalent computation requires a larger number of GPU instructions. In particular, RSL includes all standard library and DSO calls as single ops, whereas they become many machine instructions on the GPU. The sizes of our caching shaders are 28k and 22k RSL instructions for Generic Surface and Metallic Paint, respectively.

Pellacini et al. [23] describe challenges with binding overhead for the number of unique surfaces generated by specialization. Our technique has no more shaders than the original shot and our shots usually use at most a dozen unique shaders, which contrasts with the thousands of unique shaders per shot used in other studios [23]. This further emphasizes that, in our context, automatic specialization is primarily motivated by the rate at which shaders change (as well as the ability to edit surface

Configuration	RSL instr.	GPU instr.	GPU regs.
Generic Surface	19,673	<i>(combined surface/light)</i>	
spot	+1290	4653	28
point	+626	3941	24
reflection	+351	1942	20
reflection environment	+733	2721	23
ambient environment	+367	2724	22
occlusion msg	+28	863	12
Metallic Paint	22274		
spot	+1290	4461	26
“Simple” Surface	4171		
spot	+1290	3368	21

Figure 3-3: *Compiled RenderMan (RSL) vs. compiled GPU assembly instructions, and number of GPU registers. Note that the indicated total complexity of the GPU dynamic shader includes both light and surface, while RenderMan instructions are given separately.*

parameters), not their total number. Still, given increased program size limits in latest GPUs, Cg codegen could generate a single compound shader performing dynamic dispatch to subroutines implementing each surface or light. This technique is already used effectively in games.

3.4 Cache Compression

The main challenge for specialization lies in the number of values that need to be cached for large shaders. It can easily reach hundreds of scalars per deep-framebuffer element, potentially exceeding the GPU’s memory. This makes cache compression critical, and it is very effective in practice. (However, because the system is built to scale beyond GPU memory and texture size limits, using the tiling described in Chapter 5, this is only a *performance* issue and not a functionality concern.)

Static code analysis is challenging and tends to be conservative. In contrast, we find that applying simple post-processes to our final cached data provides tremendous reductions in cache complexity, sufficient to enable effective automatic deep-framebuffer generation in production scenes with a simple compiler. After caching, we analyze all channels in the deep-framebuffer and eliminate those whose

Shader	dynamic (<i>caching analysis</i>)	varying	unique (<i>compressed</i>)
generic surface	402	145	97
metallic paint	450	150	97

Figure 3-4: *The number of (scalar) values per deep-framebuffer sample for the scene in Fig. 7-2 under compression. Dynamic terms are determined by the initial caching analysis. Varying terms remain after elimination of values that are constant over the frame. Unique terms remain after further elimination of duplicated values.*

values are:

- Constant over the frame – non-*varying* terms are converted to static constants in the code.
- Identical to other channels – non-*unique* terms are replaced with references to a single common channel.

The implemented optimizations are limited. We only compress non-varying values over entire surfaces—we don’t compress uniform sub-regions of the sampled surface. We also give up some potential optimization possible in compiler analysis. For example, while the basic redundancy check works well, data-only analysis lacks the semantic information to easily detect trivial variations of some expression (e.g. x and $2 \times x$). However, it can detect values which are not semantically equivalent but which are redundant under a given evaluation, and works very well in practice.

These optimizations can reduce the number of cached components by more than a factor of 4 (Fig. 3-4). Because these optimizations inline significant new static data in the dynamic Cg shaders, this also helps the Cg compiler reduce runtime shader complexity through constant folding.

3.5 Specializing for Surface Parameters

A key advantage of automatic specialization is to allow users to selectively tweak some *surface*, as well as light parameters. When users select surface parameters as *dynamic*, the compiler can just as easily generate code with configurable surface parameters (Fig. 3-5). Many of the most commonly tuned parameters, such as gain

Editable surf. parameters	GPU instr.	regs.	relative perf.
0 (<i>baseline</i>)	3518	21	100%
18 (<i>gain</i>)	3856	27	90%
41 (<i>gain & specularity</i>)	3973	29	86%

Figure 3-5: *Preview performance as a function of the number of editable surface parameters for a variant of Generic Surface. Editing 41 scalar and vector surface parameters does not significantly slow rendering compared to light parameters alone.*

factors and specular roughness can be dynamically edited. This significantly extended the initially-planned range of application from just lighting to material look-design. In practice, the main overhead in editing surface parameters is that it requires the reevaluation of all light sources, which is costly when tens of lights are used. We have not yet implemented restricting the shading to only the modified surface samples.

Chapter 4

The Indirect Framebuffer

Traditional deep-framebuffers are pure image-space structures, which allows them to scale with image size, not scene complexity. However, because they interpret pixels as discrete surface shading samples, they cannot directly express effects where multiple shading samples contribute to a single pixel, such as antialiasing, motion blur, depth-of-field, and transparency. A direct extension would use supersampling, but this greatly increases storage and shading cost and scales poorly with the variable depth complexity introduced by transparency.

Inspired by the decoupling between shading and visibility computation central to RenderMan’s REYES pipeline, we introduce a layer of indirection between deep-framebuffer shading and visibility/display samples through a second data structure we call the *indirect framebuffer*. We first review the multisampling approach used in RenderMan before introducing our new data structure.

Background RenderMan’s REYES architecture achieves high quality and generality of antialiasing, motion blur, and depth-of-field by supersampling visibility computation, while reducing shading cost by reusing shading values rather than supersampling them [5, 3]. While smooth reconstruction of motion blur, depth-of-field, or fine geometry may require 100 or more visibility samples, the *shading rate* is commonly just roughly one shading sample per output pixel.

For this, RenderMan uses three core data structures to encode shading and visibility (Fig. 4-1.i,ii):

- Shading is performed in object space on surface shading samples called **micropolygons**.
- **Pixels** contain a uniform density of **subpixel samples**, distributed in screen-space (spatial antialiasing), time (motion blur), and aperture location (depth-of-field).
- Each subpixel sample maintains a depth-ordered **visible point list** of pointers to the micropolygons visible along that “ray”.

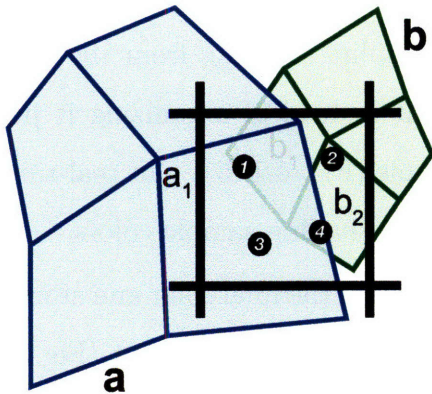
RenderMan first tessellates all primitives into micropolygons. Shaders execute over all vertices of the micropolygon grids, producing a color per vertex (Fig. 4-1.i). RenderMan then computes visibility (*hiding*) by testing each micropolygon against each subpixel sample it potentially covers (using rasterization), taking into account the aperture and time value of the sample. It performs a depth test and handles transparency by maintaining a z-ordered list of micropolygon pointers at each subpixel sample (Fig. 4-1.ii).

The color of a subpixel sample is then computed by looking up the color and opacity of each micropolygon and compositing them in depth-order. The final pixel value is the weighted average color of the subpixels, and since the subpixels are jittered in space, time, and aperture location, this achieves high quality multisampling effects while keeping shading cost tractable.

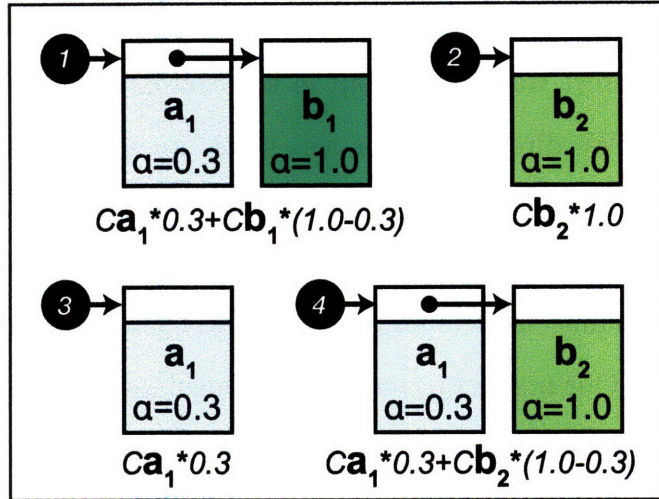
4.1 Indirect Framebuffer Data Structure

We note that each final, filtered pixel color ultimately corresponds to a simple linear combination of the shaded colors of all micropolygons visible under that pixel. Even transparency, which traditionally presents challenges due to order-dependence, ultimately factors into a single weight because we have a fixed viewing configuration. Consider the example in Fig. 4-1.ii: the first subpixel’s color is a linear combination

RenderMan / REYES

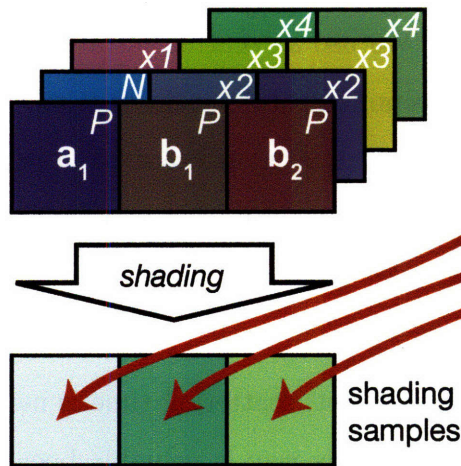


(i) Micropolygons

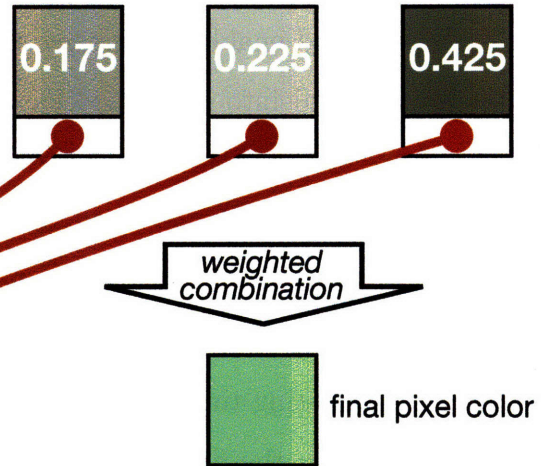


(ii) Pixel-sample Hit Lists

Lightspeed / Indirect Framebuffer



(iii) Deep-framebuffer



(iv) Indirect framebuffer

Figure 4-1: *The indirect framebuffer densely encodes variable-rate visibility information to enable efficient antialiasing and transparency under a static view. It resamples a densely-packed deep-framebuffer into screen-space to precisely reproduce RenderMan's high-quality antialiasing, but is linearized and consolidated for the given static visibility configuration, requiring far fewer unique samples for the same result.*

of shading samples a_1 and b_1 with weights given by a_1 's transparency. The final pixel value is a combination of the colors of shading samples a_1 , b_1 , and b_2 with weights 0.175, 0.225 and 0.435. When visibility is static, these cumulative linear weights similarly become static. This is similar to the principle of the direct-to-indirect transfer [11] but in the context of multisampling and transparency.

We directly exploit this static linearity while decoupling shading from the final pixel value. We use a standard deep-framebuffer, but instead of organizing it per pixel, our preprocess caches data for each shading sample (Fig. 4-1.iii). Our real-time dynamic shaders execute over this cache and output per-shading-sample colors.

Our *indirect framebuffer* encapsulates the linear nature of the final color and stores, for each pixel, a list of weights and pointers to the deep-framebuffer output (Fig. 4-1.iv). For example, the pixel in Figure 4-1.iii corresponds to three entries in the indirect framebuffer.

We need to efficiently represent the variable-length list of shading values influencing each pixel and enable progressive rendering. We use a “scatter” strategy where points are blended into each pixel location to accumulate color contribution. Each indirect framebuffer entry is encoded into a vertex array as a point, containing a pointer to a shading sample (a texture coordinate), a weight, and an output pixel coordinate (x, y) . Rendering the vertex array with blending enabled scatters the weighted colors into final pixels.

Because rendering time is dominated by shading cost, scattering time is inconsequential. Further, the fine granularity of scattering this way is valuable because it allows arbitrary progressive shading and image refinement. This is a key advantage of introducing this indirection which is not necessarily achieved by other precomputed coverage structures.

Note that one entry in the deep-framebuffer, and the resulting shaded color, often contributes to multiple neighboring pixels, especially in the presence of motion blur. This highlights the effectiveness of our decoupling (and that of RenderMan) where complex multisampling effects are achieved without scaling the cost of shading. The total number of unique shading samples can even *decrease* due to sharing between

Figure	resolution	samples	RenderMan		our approach	
			shade	subpix	shade	indir.
7-2	914x389	13x13	2.1M	32M	633k	1.6M
4-3	720x306	13x13	1.5M	21M	467k	3.8M
6-2	640x376	4x4	2.5M	2.3M	327k	716k
7-4 (α : 0.1)	720x389	8x8	54M	121M	21M	35M
7-4 (α : 0.6)	720x389	8x8	43M	58M	11M	17M
7-4 (α : 1.0)	720x389	8x8	25M	17M	3.9M	5.7M

Figure 4-2: *Original RenderMan micropolygon and pixel-sample output complexity compared to our compressed indirect framebuffer, in numbers of samples, for Figs. 4-3, 6-2, 7-2, and 7-4. Static visibility compression losslessly reduces deep-framebuffer shading samples by 3-8x relative to RenderMan’s shaded micropolygons, and reduces the number of unique indirect framebuffer samples by 3-20x relative to RenderMan’s subpixel samples.*

neighbors.

Our implementation is currently limited to static opacity. Dynamic transparency could be supported by recomputing the weights on the fly, but light-dependent transparency does not occur in our shaders. We also do not currently handle colored transparency, though it simply requires storing an RGB (instead of alpha) weight and independently blending each color channel.

4.2 Caching Visibility

During caching, we bind an additional volume shader in a special mode (vpvolume) which fires once per entry in the visible point list of each subpixel sample. This shader reads the micropolygon ID of the corresponding micropolygon at each point in the list and outputs it into a second pointcloud at the current x, y, z location. These points are projected into screen space, sorted by depth, and the micropolygon IDs correlated with those in the deep-framebuffer to reconstruct the full visible point lists of each subpixel sample.

4.3 Visibility Compression

Using the static visibility information of the indirect framebuffer, we apply two key transformations on the cached data to losslessly compress its size:

- The static linearization of the indirect framebuffer coalesces all visibility samples which reference the same shading sample at the same pixel into a single combined indirect framebuffer weight. This provides a 3-20x reduction in the size of the indirect framebuffer while producing the same output (Fig. 4-2).
- We cull all deep-framebuffer shading samples not referenced by at least one indirect framebuffer sample. We maintain a local neighborhood where necessary for derivative computation.

These optimizations reduce the number of indirect framebuffer samples by 3-20x, and the number of deep-framebuffer samples by 3-8x (Fig. 4-2), with no loss of generality, even for complex scenes involving motion blur (Fig. 4-3) and transparent hair (Fig. 7-4). This reduces not only storage size, but also computation, because shading is applied once per-deep-framebuffer sample, and resampling once per-indirect framebuffer sample. Combined with dense packing of shading values, these optimizations generally allow even heavily multisampled shots, with transparency, to require little more storage than a simple, single-sampled image-space deep-framebuffer, and to be rendered interactively.



Figure 4-3: *Lightspeed rendering from a motion-blurred RenderMan frame with 13x13 pixel samples and shading rate 1. At 720x306, RenderMan shades 1.5M micropolygons and filters 21M subpixel samples in rendering this image, while our preprocessing distills this to only 467k visible shading samples and 3.8M unique subpixel contributions to produce identical results. Shading time still significantly dominates resampling time.*

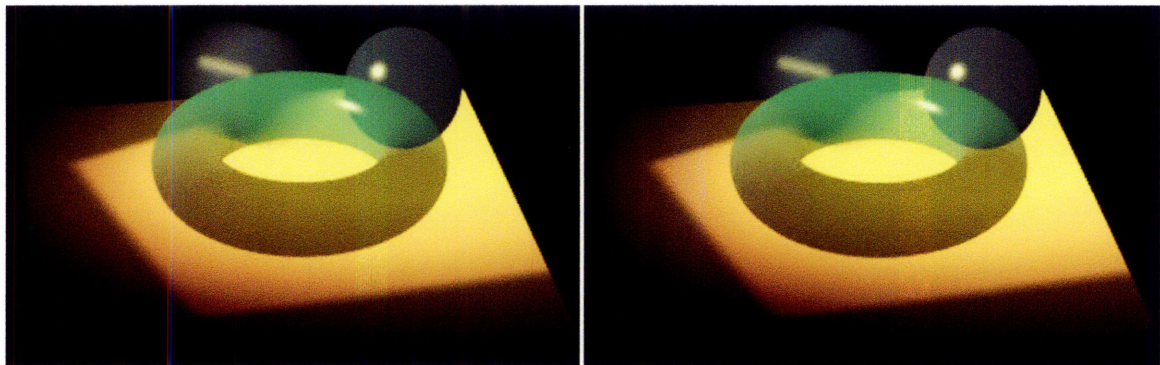


Figure 4-4: *Interacting motion blur and transparency. Left: Lightspeed. Right: RenderMan. The difference is statistically insignificant ($\ll 0.1\%$).*

Chapter 5

Scalability and Progressive Refinement

Our system must scale to final-resolution previews of massive scenes with complex shaders, while maintaining interactivity.

5.1 Tiling

High resolution previews and more complex shaders may increase cache size beyond GPU memory or texture size constraints. We divide oversized caches into screen-space tiles small enough for all hardware constraints. Each tile contains an indirect framebuffer coupled with a deep-framebuffer of all shading samples visible at those indirect framebuffer samples. We also use texture atlases because our deep-framebuffer may contain more channels than the number of bindable textures. (DirectX 10 texture arrays could increase performance by removing the arithmetic necessary to index into the deep-framebuffer atlas.)

5.2 Progressive Refinement

We rely on progressive refinement to offer both interactive feedback and slower yet faithful final image quality. We progressively refine the resolution, typically in four

steps. In the first step, we begin with 4x4 then 2x2 pixel blocks. Next, we increase to full resolution but with only one indirect framebuffer value per pixel. In the final step, we use full multisampling for the highest quality.

Each refinement stage is represented by a group of samples in our *indirect framebuffer*. We order the indirect framebuffer samples for a given pixel by weight and accumulate them progressively in passes. By simply normalizing subpixel weights for `SRC_ALPHA, ONE_MINUS_SRC_ALPHA` (instead of additive) blending, we maintain appropriate brightness. Shading is only updated for the points referenced by the indirect framebuffer samples in a given refinement batch. This also helps guarantee performance on massive scenes, because the first few refinement levels can be constrained to fit entirely on the GPU. Finally, we often disable shadows at the lowest refinement. Resampling could be performed framelessly for smoother results, but shading must still be batched for reasonable performance.

Tiles of our deep-framebuffer are stored as sets of shading samples grouped by surface type, and into batches for multiple progressive refinement passes. Passes are stored in 2D textures with arbitrary layout (2x2 quads are maintained for derivatives). In practice, shading samples are stored according to the order in which RenderMan outputs them.

5.3 Light Caching

Like prior lighting design systems, we exploit the linearity of (most) lighting by caching the contribution from all lights not currently being edited by the user. We store a light cache that gets updated when a subset of lights is temporarily “frozen.” In practice, when a light is “unfrozen”, its contribution is subtracted from the cache, and a new frozen light’s contribution is added. We retain the old parameter state with which the cache was generated to maintain correctness when subtracting. This speeds up freezing when working with tens of light sources, and has proven numerically stable over long edit sessions when using a 32-bit floating-point cache.

Changing surface parameters requires reshading the surface with all lights. In

scenes with few lights, this is still comfortably interactive. In near-final shots with dozens of lights, it may be sub-interactive, but still takes only a few seconds for useful feedback.

Light caching is complicated by the introduction of progressive refinement. Because we wish to provide initial feedback to the user as quickly as possible, it is common for the lowest refinement level of the light cache to be valid, while higher refinement levels are in various invalid states. In order to update the cache, we maintain a table of the cached light parameters for each light at every refinement level. A given cache level is valid for a light if the cached parameters match the light's current parameters. If not, the cache is updated by reshading and subtracting the contribution of the old configuration, then shading and adding the new contribution. Light configurations can be quickly compared using parameter hashes.

Chapter 6

Multipass Rendering and Management

So far, we have focused entirely on local illumination computation. However, global effects such as shadowing and translucency must also be reproduced. We first show how they can be included in our approach using multipass rendering and discuss both the necessary preprocessing and real-time components. We then address critical software architecture issues in making the development of our system tractable. The complex dependencies between multipass effects, the indirect framebuffer, and progressive refinement made it important to develop an abstraction to facilitate the inclusion of new effects and manage dependencies, as well as abstract key low-level aspects such as data-flow and bindings on the GPU.

Figure 6-1 summarizes the data-flow for our final real-time computation including shadow mapping, translucency, and indirect framebuffer effects. In this section, we explain the individual components as well as the underlying data structure, the computation graph (which is the dual of the data-flow graph since it encodes *computation* dependencies).

6.1 Shadow Mapping

Shadow mapping illustrates how multipass effects from the final rendering pipeline can be included in our architecture. Shadow maps necessitate one extra pass per light and

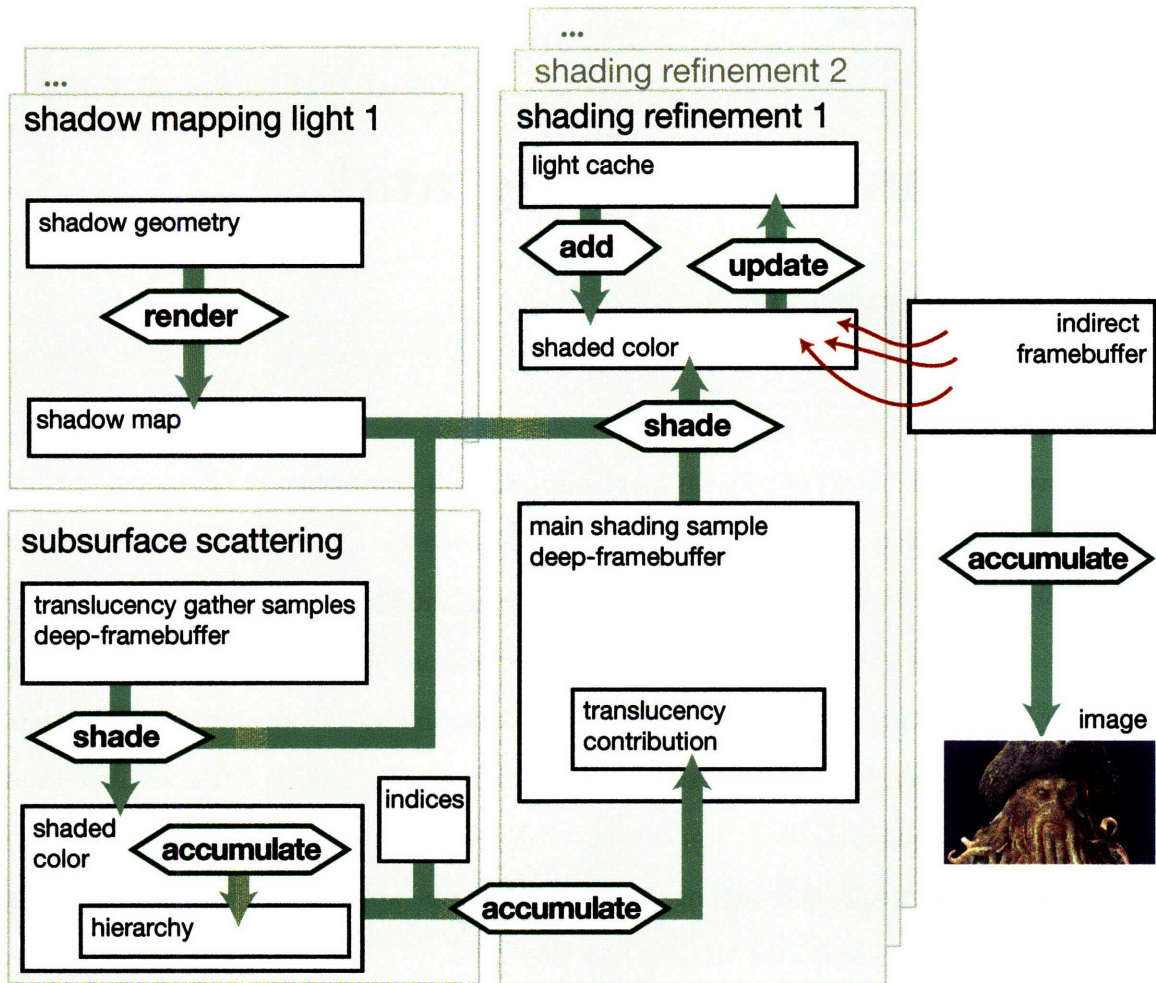


Figure 6-1: *Data-flow dependencies in multipass rendering with progressive refinement. We abstract and manage dependencies using a computation graph automatically generated for the features of a specific scene during preprocessing. Our Computation Graph data structure encodes the dependencies between cached data, shaders, and multipass outputs. It also manages communication and low-level resource management on the GPU.*

require auxiliary data from the preprocessor (scene geometry). For real-time preview, the shadow map pass communicates with the main pass through a texture and our graph interface (presented below) manages communication and dependencies when parameters are edited.

During caching, we run RenderMan a second time over the scene to extract micropolygons after all transforms and displacements are applied. We store object IDs to support selective shadow casting and receiving per-object. For specialization, RenderMan shadow mapping calls are flagged and marked dynamic. They are replaced in the dynamic code by a Cg shadow map lookup. When rendering the shadow map, we also render the object IDs to allow shadow assignments to be modified in real-time on a per-object basis. For progressive refinement, we usually disable shadowing at the coarsest resolution, similar to some modes shown by Pellacini et al. [23].

6.2 Translucency

Subsurface scattering requires the integral of incident light flux times a BSSRDF diffusion kernel over a neighborhood at each visible point. We have adapted Jensen and Buhler’s hierarchical two-pass approach [13], exactly as used in our existing offline shaders, for real-time preview. This method first creates a hierarchy of irradiance samples which enables fast hierarchical evaluation of the integral. Our scheme builds on the work by Hašan et al. [11] for indirect lighting, but instead of a wavelet approach, we directly use Jensen and Buhler’s octree hierarchy [13].

For translucency, we must distinguish the shading of visible shading samples as described in Chapter 4 from the irradiance computation at gather samples used to estimate subsurface scattering [13]. In particular, the latter cannot have view-dependent terms and usually only requires albedo and normal information. We “bake” this information during preprocessing into a separate translucency deep-framebuffer and generate a simple dynamic Cg shader, based on our offline irradiance shader, to evaluate irradiance (diffuse shading) during runtime. For each visible shading sample, we cache the indices of the set of nodes of the irradiance hierarchy that contribute

to the translucency. We also store the corresponding BSSRDF coefficient weight (the dipole kernel) [13] and distance to allow dynamic editing of the scattering depth.

For interactive preview, we first evaluate the irradiance at each gather sample using the dynamic diffuse shader and the translucency deep-framebuffer. This provides us with the leaf values of our hierarchy, stored in a texture. We then use d iterative blending passes for the d levels of the octree to accumulate the values of higher-level nodes as a sum of their children. All octree values are stored in the same texture map as the leaves.

We can then compute the color of the visible shading samples. Because only the accumulation weights, not the actual octree traversal, depend on the BSSRDF coefficients, lookups into the octree are recorded statically during preprocessing and encoded as vertex arrays, much like the indirect framebuffer. We store static BSSRDF attenuation and distance terms per-lookup, and albedo modulation per-visible-point. We then dynamically compute the BSSRDF contribution based on dynamic scattering depth (σ) values using a fragment shader, while accumulating each lookup into the hierarchy’s irradiance values using the static indices recorded during preprocessing. Note that translucency computation is performed at the granularity of shading samples and benefits from the decoupling of our indirect framebuffer, both for progressive refinement and overall efficiency.

Results Our initial results (Fig. 6-2), while promising in their fidelity, demonstrate the need for a progressive shading technique. While final scattering contributions are evaluated progressively, per visible shading point, the static octree lookups require the translucency deep-framebuffer to be completely shaded prior to any accumulation. In practice, these deep-framebuffers can be even larger than the primary deep-framebuffer—1.3M points, in this example. This means that, while the base shader renders at 2-10 Hz for initial refinement (excluding scattering computations), and changing scattering coefficients render interactively (2 Hz) for this scene, reevaluating the full subsurface scattering result takes several seconds to reach initial refinement (though subsequent refinement is very fast because the octree is already evaluated). We are considering subsampling and approximation techniques



Figure 6-2: *Subsurface scattering coefficients can be edited interactively. Top: less translucency. Bottom: more translucency. The preview renders initial refinement at 2 Hz under changing coefficients, but reshading the 1.3 million-point translucency buffer takes several seconds. The eyes contain multiple transparent layers, and appear black without the indirect framebuffer.*

for progressive refinement, but leave this to future work.

Compressing the accumulation process like Hašan et al. [11] could accelerate that process further, but accumulation is already interactive, and it would preclude dynamically editing scattering coefficients.

6.3 The Multipass Computation Graph

Multipass algorithms such as shadow mapping and translucency, together with the indirect framebuffer and progressive refinement, introduce complex data-dependencies between and computations. Furthermore, making our system extensible, and enforcing abstraction between the various components, required more care than we initially anticipated, and our original, monolithic engine quickly became challenging to maintain.

We therefore chose to abstract individual algorithms from the overall data-flow through the real-time rendering pipeline (Fig. 6-1) by using a dependency graph structure in which individual computations are encapsulated as *nodes*. Nodes communicate through *ports*, which abstract computation from dependency and data-flow, and global data-flow is encoded as edges between ports. Our core computation graph library also abstracts low-level aspects of shader and data management on the GPU, and includes a library of basic building block nodes. These nodes, and the core graph runtime, provide services including texture and framebuffer object management, caching, and the binding of Cg parameters and *interface* objects.

The graph instance for a scene is generated automatically by the compiler and preprocessing stages of our pipeline, and is used internally by the user interface application to drive the real-time renderer. In this way, the graph API also provides an interface between preprocessing, the real-time rendering engine, and the GUI application. So long as the generated graph conforms to certain basic conventions, the preprocessing stage can be updated and extended without affecting the GUI tool.

Chapter 7

Implementation and Results

Figure 7-1 summarizes our system’s fully-automatic performance on two of our shots (Figs. 7-2, 6-2). Cache sizes fit within current GPU resources, though our system scales to support out-of-core shots at much higher resolutions or with even more complex shaders.

We report all results for our current, deployed artist workstations, with dual 2.6GHz AMD Opteron 2218 processors, 8GB RAM, and NVIDIA Quadro FX 5500 (G71) graphics cards. We are generally at the limit of the capability/performance curve for our current hardware, but preliminary results suggest major performance improvements on next-generation hardware.

Our system has been integrated into the pipelines of two special effects studios. It is currently in initial release with a number of artists in production for both lighting and look-design. We have focused our efforts on ironing out the major, previously-unsolved technical challenges with such a system. As such, some technically straightforward but practically significant aspects of our implementation, such as shadow map rendering, currently lack extensive optimization, while significant effort has been paid to ensure the fidelity and scalability of the core compiler, preprocessing, and real-time shading components on complex scenes. Our subsurface scattering implementation is only a proof-of-concept and requires further optimization.

Nevertheless, initial feedback has been extremely positive. For example, artists love the freedom to experiment with complex features such as noise: “[we] usually

	Pirate (6-2)	Robot (7-2)
resolution	640x376	914x389
supersampling	4x4	13x13
lights	3	42
RenderMan (total)	409 sec	3406 sec
irradiance shading	111 sec	
material shaders	1	2
material instances	4	44
light shaders	1	5
light instances	3	42
Caching (total)	1425 sec	931 sec
initialization	8 sec	18 sec
shader specialization	24 sec	63 sec
deep-framebuffer caching	627 sec	499 sec
shadow geometry caching	105 sec	164 sec
cache compression	60 sec	187 sec
octree compression	600 sec	
Preview		
irradiance shading (1 light)	7 sec	
interaction (irradiance cached)	0.5 sec	
coarse refinement, 4x4 blocks		0.1 sec
full refinement (1 light changed)	10 sec	2.7 sec
full refinement (n lights)	29 sec (<i>3 lights</i>)	31.7 (<i>42 lights</i>)
deep-framebuffer	104 MB	256 MB
indirect framebuffer	33 MB	29 MB
irradiance deep-framebuffer	83 MB	
scattering index buffer	436 MB	

Figure 7-1: System performance compared to our RenderMan-based offline pipeline for two production shots (Figs. 7-2 & 6-2). In both, initial feedback is accelerated several orders of magnitude, to interactive rates. Caching time for Robot is significantly less than even a single offline render (common for most complex shots), because we cache with lights turned off. Caching time for the Pirate example is dominated by unoptimized octree caching and compression processes which (unnecessarily) read and write multiple GB of octree data on disk several times during caching.

shy away from noise because it takes so long to edit...this interactivity makes it much more useful.” In general, there was a strong feeling that interactive feedback not only accelerated the adjustment of key parameters (“getting that level right [previously] took me an hour!” after just tuning a light to match the background in under 10 seconds), but left users more willing to experiment aggressively. Where they previously were “timid” out of fear that, if they ran out of time during an aggressive edit, they would look bad during dailies the next morning, now they felt more free to experiment.

Caching performance Because we cache data using a modified version of the original scene and shaders run through the same offline rendering pipeline, caching performance is generally on the order of the cost of a single offline render at the same level of quality. Caching time often increases (e.g. for *Robot*), because we usually compute the cache with lights turned off, which can save significant time in shots with many lights. However, caching time for *Pirate* is dominated by an unoptimized octree preprocessing mechanism which reads and writes many gigabytes of octree data on disk several times during precomputation.

GPU vs. specialization speedup We have estimated the gain due to specialization vs. GPU execution. Since we do not have a software preview runtime, we can only perform back of the envelope calculations comparing the GPU shaders to RenderMan shaders, and prman timing with real vs. trivial shaders. Specifically, we calculate savings due to specialization by comparing the op counts for offline RSL and specialized GPU shaders, counting the sum of light and surface shaders given in figure 3-3. We measure RenderMan’s non-shading overhead (scene setup, tessellation, displacement, and visibility computation) by subtracting the time to render our scenes with trivial no-op shaders from the same scenes with full shaders and lights demonstrated in our results. For the included scenes, we estimate that specialization and caching provide a 100x speedup while execution on the GPU brings another 20x. The coarsest level of refinement provides an extra 10-100x.

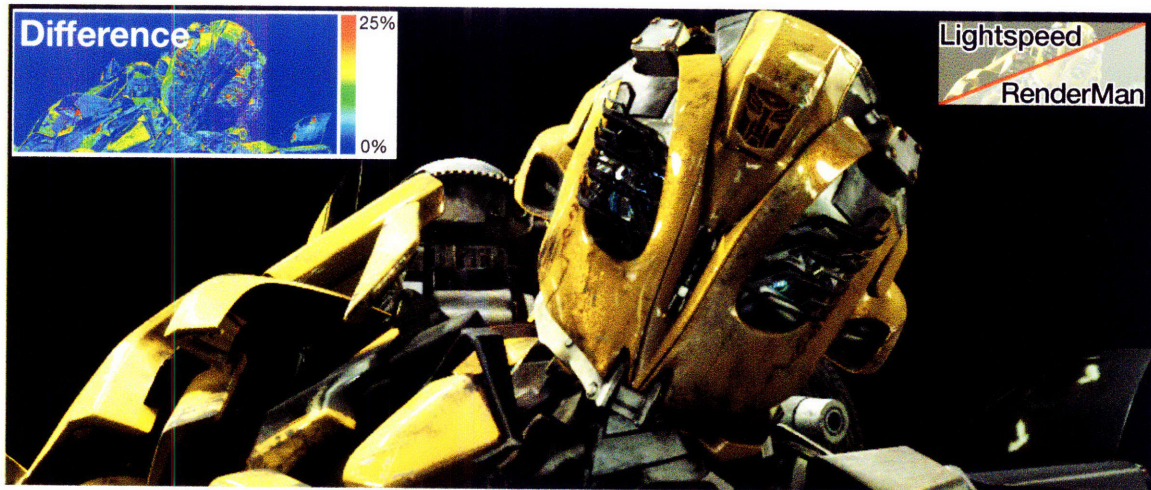


Figure 7-2: An automatically-generated preview at 914x389 resolution with 13x13 supersampling for a scene featuring 42 spot, environment, and message-passing lights and multiple 20k instruction surface shaders. The upper-left half of the image is rendered with our approach while the lower right is the final RenderMan frame – the seam is barely visible. The error heat map is in percentage of maximum 8-bit pixel value and is mostly due to shadow map artifacts. This scene renders interactively at 4x4 subsampled resolution at 9.2 Hz, while refining to the above antialiased final-quality in 2.7 seconds, compared to 57 minutes in RenderMan.

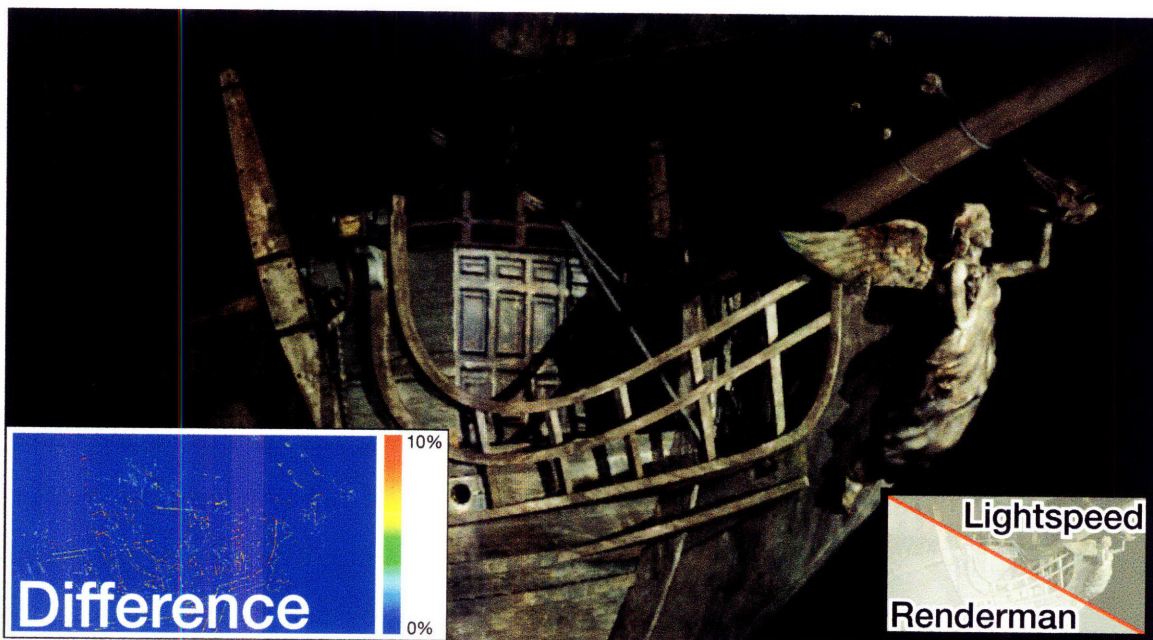


Figure 7-3: The upper-right half of the image is rendered with our approach while the lower left is the final RenderMan frame. Initial refinement renders at over 20 Hz with our full 4k instruction specialized surface shader and spot light, including shadows. Error is in percentage of max pixel value.



Figure 7-4: *430k transparent hairs ($\alpha = 0.6$, opacity threshold: 0.96) rendered at 720x389 with 8x8 sampling. This generates 43M micropolygons and 58M pixel samples in RenderMan, and condenses to 11M visible shading samples and 17M unique visibility samples through lossless visibility compression, rendering at 12 Hz and fully refining in 33 secs. Compression and performance are even better at $\alpha = 1.0$, but $\alpha = 0.1$ (threshold: 0.996) generates 21M visible shading samples, overflowing the 16M sample textures we currently use (cf. Fig. 4-2).*

7.1 Scalability

Shadow geometry scales with scene complexity and is the main scalability limitation, in practice. Using micropolygons instead of source primitives was a design decision to avoid re-implementing displacement and every primitive supported by prman. We control shadow-geometry level of detail by altering the shading rate of the shadow bake pass. Additional mesh decimation passes could be useful.

Aside from shadowing, our system effectively scales with image complexity. The indirect framebuffer and cache compression dramatically reduce memory costs. Transparency is the main difference from previous techniques because it adds an unbounded number of samples. We created a complex scene to test scalability (Fig. 7-4): 430k transparent hair fibers ($\alpha = 0.1$, opacity threshold= 0.996), resulting in 55M prman micropolygons and 20M visible Lightspeed shading samples rendered at

720x389 with 64x supersampling. This overflows our shade sample texture because of the GPU's 4k×4k (16M) texture limit. However, with α reduced to 0.6, the same scene only requires 11M shade samples (vs. 43M in prman) and works at 12 Hz (33 secs for full refinement because the full cache is 2GB and needs to be paged). With no transparency, Lightspeed shades just 4M samples (vs. 25M for prman) at 22 Hz (5.5 secs for full refinement). The 16M limit can trivially be increased by using multiple textures or 8k textures in DirectX 10.

For our production scenes, however, we have not encountered such extreme cases. Our artists avoid transparent hair in favor of smaller sub-pixel hair because these same scalability problems apply in prman. In fact, though unbounded, transparency consistently contributes much less to total frame complexity than (bounded) multisampling in our scenes.

While the worst case scales with supersampled image complexity (times depth complexity for transparency), the key goal of our design—visibility compression and the linearization of visibility into the indirect framebuffer—is to provide real-world scaling much closer to pixel-complexity, even with motion blur (Fig. 4-3), sub-pixel microgeometry like hair (Fig. 7-4), and a modest average transparency depth.

The overall conclusion of our tests, ignoring shadowing, is:

- We can handle a lot of fine geometry, *or* probably handle a lot of very transparent coarse geometry, but our current implementation will not handle a lot of very transparent *and* fine geometry that completely fills the image, with antialiasing.
- We can handle a lot of fine geometry that is semi-transparent even if it fills the image, with high antialiasing.

Where scene complexity can become an issue for the indirect framebuffer is during caching. Because simple methods of caching (`bake3d`) extract all shaded grids from prman, initial cache sizes can be very large, and compression becomes disk i/o bound. We addressed this by pushing compression in-memory with the renderer during caching (as a DSO), which greatly accelerates caching and culling.

The number of unique shaders can also be an issue. However, if a given surface

shader is used for multiple surfaces with different parameters, we only need to specialize it once. The total number of dynamic shaders is the product of the number of different light shaders and the number of surface shaders (not the number of instances). Because we mostly use übershaders, this is not a problem for our workloads (≤ 10 -100 combinations in practice, Fig. 7-1), though it would be for studios with thousands of unique shaders in a shot. This might be addressed with established techniques, as discussed in Footnote 1.

7.2 Challenges and Limitations

In practice we find our approach quite robust. Major challenges we have addressed include:

- Automatically specialized shaders fit within current GPU limits. Future shaders will surpass the limits of our current hardware, but newer GPUs have already elevated the relevant program and register size limits by at least an order of magnitude.
- Dynamic calls to external C routines are largely eliminated during specialization, and, where they aren't, they have been effectively emulated on the GPU or made *cache-required*.
- Generated deep-framebuffers are compressed to modest sizes, even for our more complicated scenes and shaders.
- GPU texture limits are abstracted through tiling.
- Complex visibility is effectively compressed, even at high multisampling rates.
- Interactivity is maintained in the face of complexity by progressive refinement.

Our key limitations are the same faced by any GPU shading system—namely, that operations not easily expressed as native GPU instructions require special handling. Most importantly, non-local shading must be handled explicitly using multipass algorithms. We have achieved this for shadows and translucency, but additional implementation is required for other effects.

Still, a number of features cannot be translated and would result in an error message if deemed dynamic. Fortunately, most such features are usually not used in the dynamic parts of shaders in our studio. This may not be true in all studios.

Ray Tracing We do not perform ray casting. Note that specular ray tracing could be previewed in a deep-framebuffer using indirect buffers (ray intersections do not change unless the index of refraction is edited for transmitted rays). This is future work. The main limitation concerns ray-casting for shadows and inter-reflections.

Ambient occlusion Lightspeed would require re-caching of occlusion if object-object occlusion assignments changed. Our artists only edit occlusion gain during lighting design, and inter-object occlusion, itself, can be cached.

Shadows Our system currently does not implement deep shadows and this is a serious limitation for scenes with hair. Similarly, we do not implement area shadows.

Brickmaps and pointclouds Memory management would present challenges for implementing brickmaps. We do not support them in dynamic code. This is a particular problem if brickmaps are used in a light shader. Our subsurface scattering implementation is an example where a point cloud is statically sampled through a complex data structure at cache time, but the returned values are dynamic.

Non-linear lights Non-linear contributions are not easily cached.

Dynamic loops Dynamic loops containing cached expressions are a limitation. We support them in the special case where they are bounded, since we statically allocate space in the deep-framebuffer. Figure 6-2 uses bounded dynamic loops for layered materials.

Chapter 8

Conclusions and Future Work

We have introduced a system for the real-time preview of RenderMan scenes during lighting design. Our method automatically specializes shaders into a static RenderMan pass that generates a deep-framebuffer, and a dynamic Cg pass that uses the deep-framebuffer to enable real-time preview on a GPU. Cache compression enables automatically generated deep-framebuffers to fit in modest GPU memory for complex production shots. We have introduced the *indirect framebuffer* which efficiently encodes multisampling for high-quality rendering with transparency and motion blur. Our computation graph-based system architecture is flexible and is amenable to multipass rendering algorithms and progressive refinement, which we demonstrate with shadow mapping and subsurface scattering.

We were surprised by the effectiveness of cache compression. Initially, we assumed we would build complex compiler analyses to control cache size. However, due to the data-parallel nature of shading, redundancy abounds, and simple post-processes easily uncover savings which static analysis could not recognize.

As a whole, our system brings a level of automation that greatly simplifies interactive lighting preview and alleviates the need to write and maintain different shaders for final rendering, preprocessing, and preview. However, it does not close the debate between manual instrumentation and automatic specialization. The manual programming of preview shaders can bring an extra level of flexibility, in particular to adapt the level of detail to further accelerate preview, as illustrated in *lpics*

[23], though Pellacini separately showed that automatic level-of-detail can help [22]. In the long run, we believe that lighting preview should be addressed in a way similar to traditional programming: automatic tools are provided for compilation and optimization, and the programmer can provide hints or manually optimize and simplify critical portions of the code based on profiling tools.

Our specialization is not restricted to a fixed viewpoint. We have effectively specialized shaders against a dynamic view direction. Using 3D point rendering of the shaded deep-framebuffer has produced promising initial results with dynamic view direction.

Still, the greatest limitation to deep-framebuffer rendering is its basis in local shading. As global illumination becomes prevalent in production rendering, the ability to integrate global effects into this system will determine its future success. Fortunately, our techniques are also not specific to GPUs. Rather, they are generally useful for reducing complex shading to efficient data-parallel execution, including on future manycore CPUs, and this may ultimately be the avenue through which global effects such as ray tracing, which greatly benefit from adaptive data structures, are most efficiently achieved.

Bibliography

- [1] Alias. Interactive photorealistic rendering, 1999.
- [2] 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.
- [3] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: creating CGI for motion pictures*. Morgan Kaufmann, 2000.
- [4] Avi Bleiweiss and Arcot Preetham. Ashli—Advanced shading language interface. *ACM SIGGRAPH Course Notes*, 2003.
- [5] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102, July 1987.
- [6] Julie Dorsey, James Arvo, and Donald Greenberg. Interactive design of complex time dependent lighting. *IEEE Computer Graphics & Applications*, 15(2):26–36, March 1995.
- [7] Reid Gershbein and Patrick M. Hanrahan. A fast relighting engine for interactive cinematic lighting design. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 353–358, July 2000.
- [8] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 343–350, August 1995.
- [9] Pat Hanrahan. Ray tracing algebraic surfaces. In *Proc. of SIGGRAPH 1983*, pages 83–90, 1983.
- [10] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proc. of SIGGRAPH 1990*, pages 289–298, 1990.
- [11] Miloš Hašan, Fabio Pellacini, and Kavita Bala. Direct-to-indirect transfer for cinematic relighting. *ACM Transactions on Graphics*, 25(3):1089–1097, July 2006.

- [12] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
- [13] Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics*, 21(3):576–581, July 2002.
- [14] Thouis R. Jones, Ronald N. Perry, and Michael Callahan. Shadermaps: a method for accelerating procedural shading. Technical report, Mitsubishi Electric Research Laboratory, 2000.
- [15] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proc. of SIGPLAN 1996*, pages 215–225, 1996.
- [16] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, July 2003.
- [17] T. Mogensen. The application of partial evaluation to ray-tracing. Master’s thesis, DIKU, U. of Copenhagen, Denmark, 1986.
- [18] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics*, 22(3):376–381, July 2003.
- [19] Nvidia. Sorbetto relighting technology, 2005.
- [20] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 159–168, July 1998.
- [21] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 425–432, July 2000.
- [22] Fabio Pellacini. User-configurable automatic shader simplification. *ACM Transactions on Graphics*, 24(3):445–452, August 2005.
- [23] Fabio Pellacini, Kiril Vidimčević, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics*, 24(3):464–470, August 2005.
- [24] Pixar. Irma, 2001.
- [25] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural data flow analysis via graph reachability. In *Proc. of SPPL 1995*, pages 49–61, 1995.

- [26] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 197–206, August 1990.
- [27] Carlo H. Séquin and Eliot K. Smyrl. Parameterized ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, pages 307–314, July 1989.
- [28] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics*, 21(3):527–536, July 2002.
- [29] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics*, 23(3):469–476, August 2004.
- [30] Daniel Wexler, Larry Gritz, Eric Enderton, and Jonathan Rice. Gpu-accelerated high-quality hidden surface removal. In *Graphics Hardware 2005*, pages 7–14, July 2005.