The Lightspeed Automatic Interactive Lighting Preview System

Jonathan Ragan-Kelley* Charlie Kilpatrick† Brian W. Smith† Doug Epps‡ Paul Green* Christophe Hery† Frédo Durand*

*MIT CSAIL †Industrial Light & Magic ‡Tippett Studio

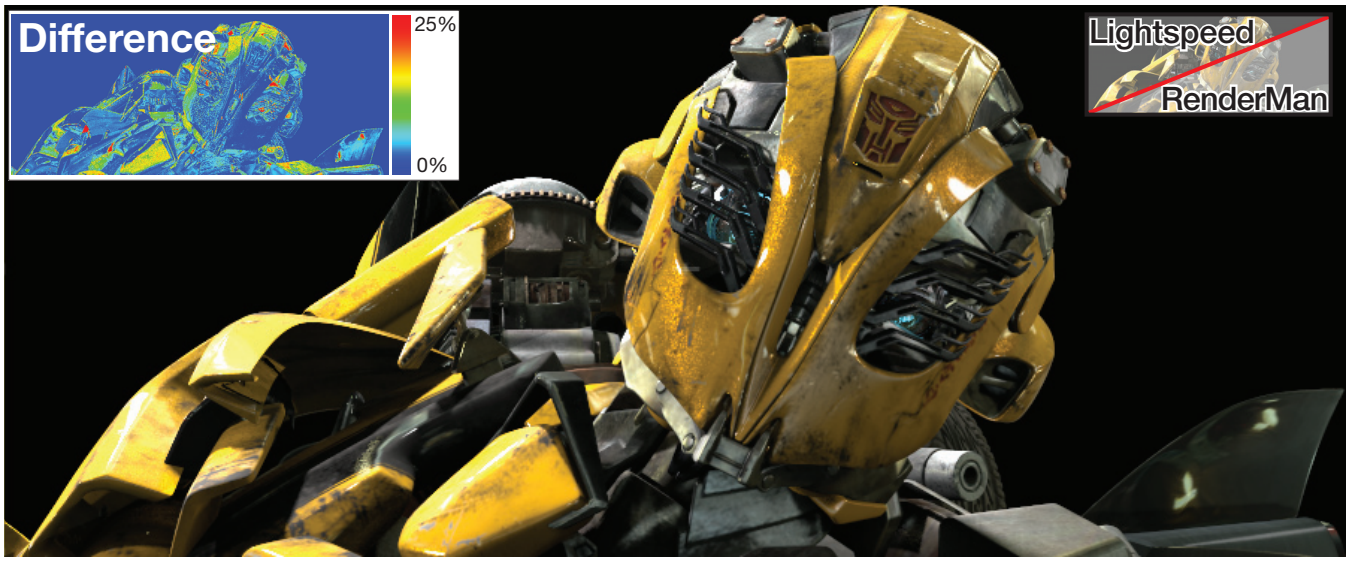

Figure 1: *An automatically-generated preview at 914x389 resolution with 13x13 supersampling for a scene featuring 42 spot, environment, and message-passing lights and multiple 20k instruction surface shaders. The upper-left half of the image is rendered with our approach while the lower right is the 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.*

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.

Keywords: Lighting Preview, Interactive Rendering, Data-flow Analysis, RenderMan, Programmable Shading, GPUs

1 Introduction

Configuring lights is a critical bottleneck in modern production rendering, and recent advances have sought to provide real-time preview using deep-framebuffers and graphics hardware [Gershbein and Hanrahan 2000; Pellacini et al. 2005]. 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, these approaches require substantial additional work from shader authors. For example, in the lpics system deployed at Pixar [Pellacini et al. 2005], 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.

We alleviate the need to author multiple versions of a shader by automatically translating unmodified 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 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.

We describe a full production relighting system that is being deployed in two studios with different rendering workflows.

1.1 Prior Work

Fast relighting has long been a major area of research [Dorsey et al. 1995; Ng et al. 2003]. 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 [Alias 1999; Pixar 2001; Nvidia 2005; Tabellion and Lamorlette 2004]. However, such optimizations must be integrated at the core of a system and are still far from interactive for film scenes.

Séquin and Smyrl [1989] 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 [2000] which cached intermediate results in a deep-framebuffer inspired by G-Buffers [Saito and Takahashi 1990]. They cached a fixed set of data, and approximated shading with multitexturing. Pellacini et al. performed shading on programmable graphics hardware [2005] 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 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 [2005], 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 [Hašan et al. 2006]. 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 [2000]. However, their technique only cached shading computation—not tessellation, displacement, etc.—and required manual shader segmentation.

Our goals cannot be fully met by pre-computed radiance transfer (PRT) techniques [Sloan et al. 2002; Ng et al. 2003], because they usually make assumptions on the reflectance or lighting and have significant precomputation cost. In contrast, we need to handle the effect of local point light sources and arbitrary reflectance. Furthermore, computing illumination itself is a large part of our run-time calculation as production light shaders are quite complex.

Compiler specialization of graphics computation was first used for ray tracing [Hanrahan 1983; Mogensen 1986; Andersen 1996]. 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 [1995; 1996]. 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 [1996]. 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.

Peercy et al. [2000] and Bleiweiss and Preetham [2003] 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.

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 have more complexity and could prove harder to fully map to graphics hardware.

However, it later became apparent that our approach can also enable the modification of many, but not all, material appearance parameters, and 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.
- **Fast initial precomputation** – To be accepted by artists, this tool should not make it take longer 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** – Though 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 final rendering, including antialiasing, motion blur, and transparency.
- Using the **same workflow** – in particular the same light editing GUI, which varies from studio to studio. This requires our system to communicate with different GUI software.

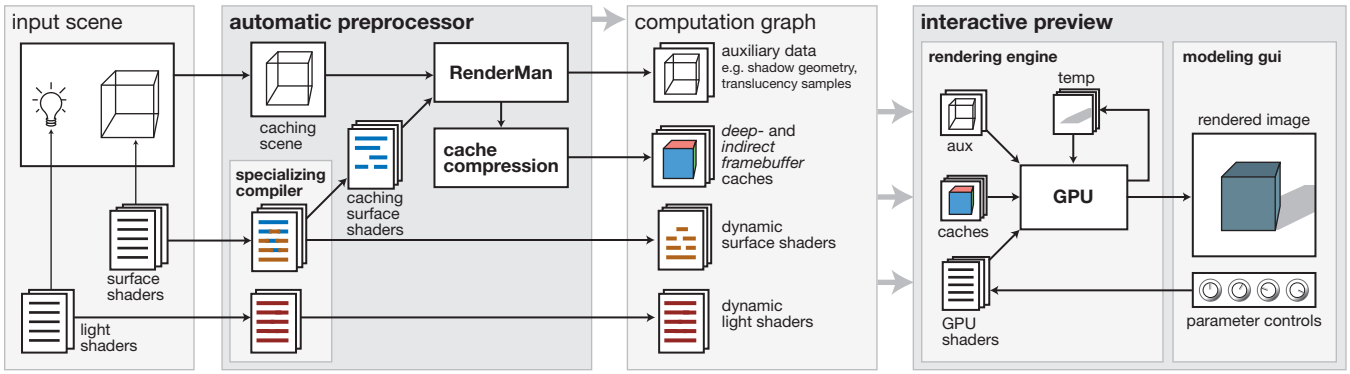

Figure 2: *Our system takes as input the original RenderMan scene with its shaders. Our* specializing compiler *automatically separates a shader into static and dynamic parts and uses RenderMan to cache static computation and auxiliary data. The dynamic part is translated into Cg.* Cache compression *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 engine executes the graph to generate intermediate data (shadow maps, etc.) and run the dynamic shaders over the cache on the GPU. A new* indirect framebuffer *enables antialiasing and transparency. The GUI application modifies light parameters through the graph interface.*

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) can be decomposed into an automatic preprocess and a run-time 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 be expected to yield a performance penalty for the interactive preview compared to manually optimized and simplified code [Gershbein and Hanrahan 2000; Pellacini et al. 2005], 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 not been an issue.

Indirect framebuffer Our core real-time rendering 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 an *indirect framebuffer* to decouple shading samples from final pixel values, thereby efficiently handling antialiasing, motion blur, and transparency. It also enables progressive refinement (Sec. 4, 5).

Cache 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 over 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 also output 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 dependences between different elements of 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.

3 Automatic Deep-Framebuffer Caching

We wish to automatically generate a deep-framebuffer and real-time preview. We first need to 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.

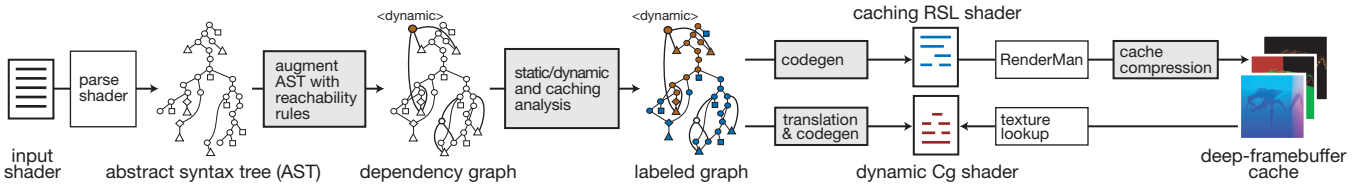

Figure 3: Specializing compiler. *The input shader is represented as an abstract syntax tree (AST). We augment it to encode dependency between variables and expressions. To decide if an expression is dynamic, we query whether it depends on any dynamic parameters. Once the shader has been split, we generate two new shaders, a caching shader and a real-time shader. 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.*

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 [Horwitz et al. 1990; Reps et al. 1995]. 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 respect to light parameters.

Dependence analysis The first step of our analysis (Fig. 3) adds global dependencies to transform an abstract syntax tree (AST) representation of the shader into a *dependency graph* that encodes all dependencies between expressions. We add a *dynamic* node and connect it to the dynamic parameters, specified by name. We then simply query whether each expression depends on a dynamic parameter by testing if it can reach the dynamic node. The core global dependency rules are described separately [Ragan-Kelley 2007]. We perform dead-code elimination using the same dependence graph by connecting output values to a new *output* node.

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.

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.

RenderMan precomputation Caching computations are emitted as a new RSL shader. 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. 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).

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 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, enabling 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 for large texture filter kernels. These primarily depend on surface partial derivatives, which are not dynamic, so we simply cache them when necessary.

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* [Mark et al. 2003].

This approach can only automatically translate light shaders which do not rely on *cache-required* functionality—namely, external C calls. 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 do not find this problematic.

3.3 Specialization Results

Figure 4 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. RSL instructions tend to be higher-level, and the equivalent computation requires a larger number of GPU instructions. The sizes of our caching shaders are 28k and 22k RSL instructions for Generic Surface and Metallic Paint, respectively.

Pellacini et al. [2005] 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 [Pellacini et al. 2005][1]. 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 parameters), not their total number.

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, as well as the tiling described in Section 5, critical.

[1]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.

Configuration	RSL instr.	GPU instr.	GPU regs.
Generic Surface	19,673	*(combined surface/light)*	
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 4: *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.*

3.4 Cache Compression

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 with a simple compiler. After caching, we analyze all channels in the deep-framebuffer and eliminate those whose 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.

These optimizations can reduce the number of cached components by more than a factor of 4 (Fig. 5). 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.

Shader	dynamic (*caching analysis*)	varying	**unique** (*compressed*)
generic surface	402	145	**97**
metallic paint	450	150	**97**

Figure 5: *The number of (scalar) values per deep-framebuffer sample for the scene in Fig. 1 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.*

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. 6). Many of the most commonly tuned parameters, such as gain factors and specular roughness can be dynamically edited. This significantly extended the initially-planned range from lighting to look-design. In practice, the main overhead in editing surface parameters is that it requires the reevaluation of all light sources.

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

Figure 6: *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.*

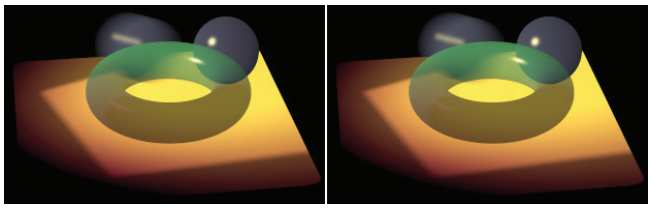

Figure 7: *Motion blur and transparency. Left: Lightspeed. Right: RenderMan. The difference is statistically insignificant ($\ll 0.1\%$).*

Figure 8: *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.*

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 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 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 [Cook et al. 1987; Apodaca and Gritz 2000]. 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. 9.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, pro-

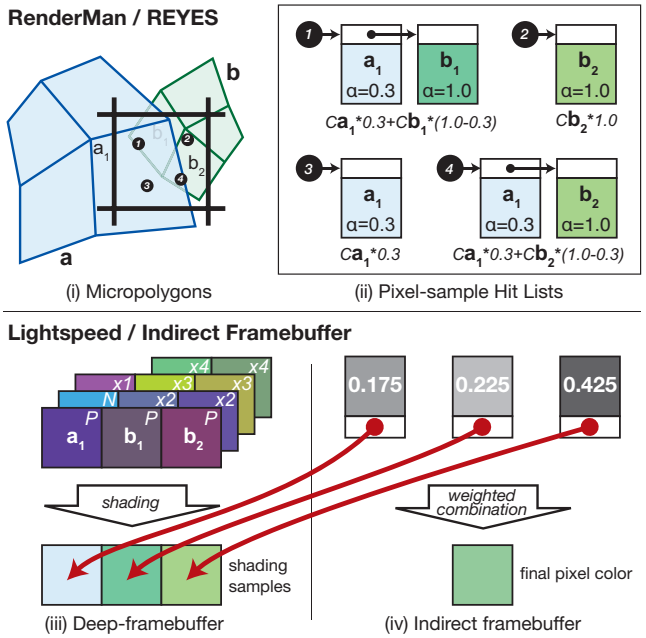

Figure 9: *The* indirect framebuffer *densely encodes variable-rate visibility information to enable efficient antialiasing and transparency under a static view. It resamples a densely-packed deep-framebuffer into screen-space to precisely reproduce RenderMan's high-quality antialiasing, but is linearized and consolidated for the given static visibility configuration, requiring far fewer unique samples for the same result.*

ducing a color per vertex (Fig. 9.i). RenderMan then computes visibility (*hiding*) by testing each micropolygon against each subpixel sample it potentially covers (rasterization), taking into account the aperture and time value of the sample. A depth test is performed and transparency is handled by maintaining a z-ordered list of micropolygon pointers at each subpixel sample (Fig. 9.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 assume a fixed viewing configuration. Consider the example in Fig. 9.ii: the first subpixel's color is a linear combination 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 [Hašan et al. 2006] but in the context of multisampling and transparency.

We directly exploit this static linearity while decoupling shading and final pixel value. We first use a standard deep framebuffer, but instead of organizing it per pixel, our preprocess caches data for each shading sample (Fig. 9.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. 9.iv). For example, the pixel in Figure 9.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 rendered at 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.

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.

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 weight and independently blending each color channel.

Figure	resolution	samples	RenderMan shade	RenderMan subpix	our approach shade	our approach indir.
1	914x389	13x13	2.1M	32M	633k	1.6M
8	720x306	13x13	1.5M	21M	467k	3.8M
12	640x376	4x4	2.5M	2.3M	327k	716k
15 (α: 0.1)	720x389	8x8	54M	121M	21M	35M
15 (α: 0.6)	720x389	8x8	43M	58M	11M	17M
15 (α: 1.0)	720x389	8x8	25M	17M	3.9M	5.7M

Figure 10: *Original RenderMan micropolygon and pixel-sample output complexity compared to our compressed indirect framebuffer, in numbers of samples, for Figs. 1, 8, 15, and 12. 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.*

4.2 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. 10).
- 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. 10), with no loss of generality, even for complex scenes involving motion blur (Fig. 8) and transparent hair (Fig. 15). 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.

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. 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.

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 3 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 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.

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 multiple 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 significantly 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.

6 Multipass Rendering and Management

So far, we have focused on purely 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 dependences 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 dependences, as well as abstract key low-level aspects such as data-flow and bindings on the GPU.

Fig. 11 summarizes the data-flow for our final real-time computation including shadow mapping, translucency, and indirect framebuffer effects.

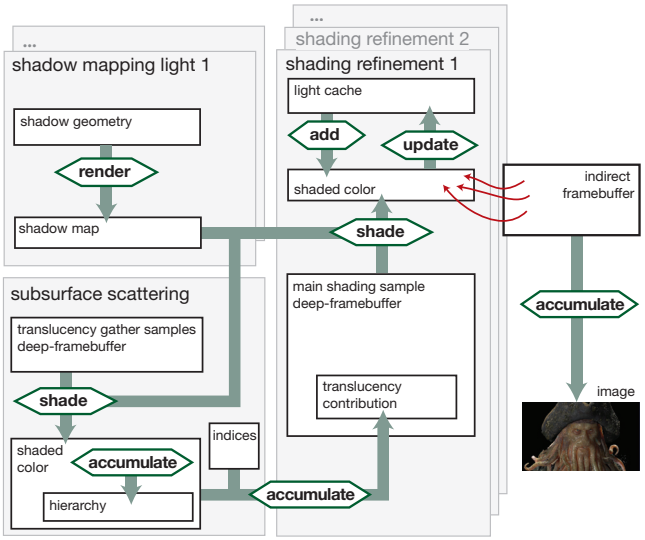

Figure 11: *Data-flow dependencies in multipass rendering with progressive refinement. We abstract and manage dependencies using a computation graph* automatically generated *for the features of a specific scene during preprocessing.*

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 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 dependences 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.

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 [2002], 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. [2006] for indirect lighting, but instead of a wavelet approach, we directly use Jensen and Buhler's octree hierarchy [2002].

For translucency, we must distinguish the shading of visible shading samples as described in Section 4 and the irradiance computation at gather samples used to estimate subsurface scattering [Jensen and Buhler 2002]. 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) [Jensen and Buhler 2002] 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 instead 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 (sigma) 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. 12), 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 changing scattering coefficients render interactively (2 Hz) for this scene, and the base shader renders at 2-10 Hz for initial refinement, excluding scattering computations, reevaluating the 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 for progressive refinement, but leave this to future work.

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.

Figure 12: *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.*

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. 11) 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.

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.

7 Implementation and Results

Figure 13 summarizes our system's fully-automatic performance on two of our shots (Figs. 1, 12). 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. 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.

	Pirate (12)	**Robot (1)**
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 (*3 lights*)	31.7 (*42 lights*)
deep-framebuffer	104 MB	256 MB
indirect framebuffer	33 MB	29 MB
irradiance deep-framebuffer	83 MB	
scattering index buffer	436 MB	

Figure 13: *System performance compared to our RenderMan-based offline pipeline for two production shots (Figs. 1 & 12). 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.*

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 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. Subsurface scattering is only 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 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.

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. 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.

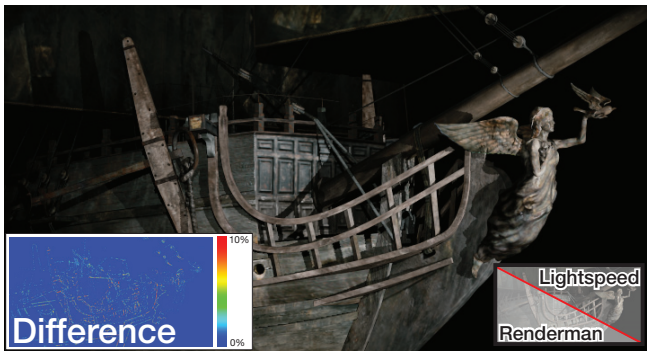

Figure 14: *The upper-right half of the image is rendered with our approach while the lower left is the final RenderMan frame. Initial refinement renders at over 20 Hz with our full 4k instruction specialized surface shader and spot light, including shadows. Error is in percentage of max pixel value.*

Figure 15: *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. 10).*

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 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. 15): 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 4kx4k (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. 8), sub-pixel microgeometry like hair (Fig. 15), and a modest average transparency depth.

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

- We can handle a lot of fine geometry, *or* 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 (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 ($\leq$10-100 combinations in practice, Fig. 13), 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:

- 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.
- 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.

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, 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 shadowing 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.

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 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 12 uses bounded dynamic loops for layered materials.

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, 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 [Pellacini et al. 2005], though Pellacini separately showed that automatic level-of-detail can help [2005]. 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.

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 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 are most efficiently achieved.

Acknowledgments Numerous 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.

References

ALIAS, 1999. Interactive photorealistic rendering.

ANDERSEN, P. H. 1996. Partial evaluation applied to ray tracing. In *Software Engineering in Scientific Computing*, Vieweg, W. Mackens and S. Rump, Eds., 78–85.

APODACA, A. A., AND GRITZ, L. 2000. *Advanced RenderMan: creating CGI for motion pictures*. Morgan Kaufmann.

BLEIWEISS, A., AND PREETHAM, A. 2003. Ashli—Advanced shading language interface. *ACM SIGGRAPH Course Notes*.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, 95–102.

DORSEY, J., ARVO, J., AND GREENBERG, D. 1995. Interactive design of complex time dependent lighting. *IEEE Computer Graphics & Applications 15*, 2 (Mar.), 26–36.

GERSHBEIN, R., AND HANRAHAN, P. M. 2000. A fast relighting engine for interactive cinematic lighting design. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 353–358.

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, 343–350.

HANRAHAN, P. 1983. Ray tracing algebraic surfaces. In *Proc. of SIGGRAPH 1983*, 83–90.

HAŠAN, M., PELLACINI, F., AND BALA, K. 2006. Direct-to-indirect transfer for cinematic relighting. *ACM Transactions on Graphics 25*, 3 (July), 1089–1097.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1, 26–60.

JENSEN, H. W., AND BUHLER, J. 2002. A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics 21*, 3 (July), 576–581.

JONES, T. R., PERRY, R. N., AND CALLAHAN, M. 2000. Shadermaps: a method for accelerating procedural shading. Tech. rep., Mitsubishi Electric Research Laboratory.

KNOBLOCK, T. B., AND RUF, E. 1996. Data specialization. In *Proc. of SIGPLAN 1996*, 215–225.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics 22*, 3 (July), 896–907.

MOGENSEN, T. 1986. *The application of partial evaluation to ray-tracing*. Master's thesis, DIKU, U. of Copenhagen, Denmark.

NG, R., RAMAMOORTHI, R., AND HANRAHAN, P. 2003. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics 22*, 3 (July), 376–381.

NVIDIA, 2005. Sorbetto relighting technology.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 425–432.

PELLACINI, F., VIDIMČE, K., LEFOHN, A., MOHR, A., LEONE, M., AND WARREN, J. 2005. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics 24*, 3 (Aug.), 464–470.

PELLACINI, F. 2005. User-configurable automatic shader simplification. *ACM Transactions on Graphics 24*, 3 (Aug.), 445–452.

PIXAR, 2001. Irma.

RAGAN-KELLEY, J. 2007. *The Lightspeed Automatic Interactive Lighting Preview System*. Master's thesis, Massachusetts Institute of Technology.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural data flow analysis via graph reachability. In *Proc. of SPPL 1995*, 49–61.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, 197–206.

SÉQUIN, C. H., AND SMYRL, E. K. 1989. Parameterized ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, 307–314.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*, 3 (July), 527–536.

TABELLION, E., AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics 23*, 3 (Aug.), 469–476.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. Gpu-accelerated high-quality hidden surface removal. In *Graphics Hardware 2005*, 7–14.