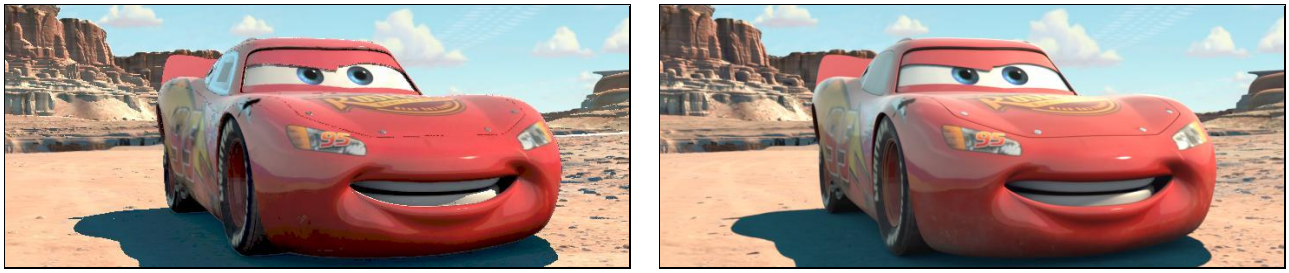
Lpics: a Hybrid Hardware-Accelerated Relighting Engine for Computer Cinematography

Fabio Pellacini* Kiril Vidimče† Aaron Lefohn† Alex Mohr† Mark Leone† John Warren†
Pixar Animation Studios



Lpics render ≈ 0.1s Final render ≈ 2000s

Figure 1: Images rendered by Lpics relighting engine versus software renderer.
Times reported are the time a lighting artist must wait for feedback after moving one light.

Abstract

In computer cinematography, the process of lighting design involves placing and configuring lights to define the visual appearance of environments and to enhance story elements. This process is labor intensive and time consuming, primarily because lighting artists receive poor feedback from existing tools: interactive previews have very poor quality, while final-quality images often take hours to render.

This paper presents an interactive cinematic lighting system used in the production of computer-animated feature films containing environments of very high complexity, in which surface and light appearances are described using procedural RenderMan shaders. Our system provides lighting artists with high-quality previews at interactive framerates with only small approximations compared to the final rendered images. This is accomplished by combining numerical estimation of surface response, image-space caching, deferred shading, and the computational power of modern graphics hardware.

Our system has been successfully used in the production of two feature-length animated films, dramatically accelerating lighting tasks. In our experience interactivity fundamentally changes an artist's workflow, improving both productivity and artistic expressiveness.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.2 [Computer Graphics]: Graphics Systems.

Keywords: Interactive Rendering, Relighting, GPUs

*e-mail: fabio@graphics.cornell.edu
†e-mail:{vkire,lefohn,amohr,mleone,jwarren}@pixar.com

1 Introduction

High-quality lighting in computer-animated films is labor intensive and time consuming: for example, one recent computer-animated feature, Pixar's *The Incredibles*, required a crew of 40 lighting artists. These artists need fast and accurate feedback, but dramatic increases in shot complexity have forced them to either endure long render times or accept drastic reductions in image quality in preview renders, which often employ lower sampling rates, stand-in models, and simplified surface shaders. In this paper we describe a system called *lpics* that allows artists to *interactively* configure and refine lights in scenes of extremely high complexity with minimal approximations to the final rendered appearance.

1.1 System summary

Our system is a relighting engine for scenes in which surface and light shading is specified by RenderMan shaders [Pix 2000]. Its basic operation is quite simple (similar systems are discussed in Section 1.3):

- Surface shaders are instrumented with calls that generate *image-space caches* containing the parameters of an illumination model, such as the displaced point and normal, calculated surface color, etc. Caching samples on a per-pixel basis allows scenes of extremely high geometric complexity to be represented compactly and re-rendered efficiently.
- A conventional software renderer executes the instrumented surface shaders and generate caches, which need to be updated when geometric data or surface shaders changes.
- Light shaders are *simplified* and *translated* into a hardware shading language (such as Cg [Mark et al. 2003]). Currently this simplification is performed manually, incorporating sampling approximations (using fewer samples) and interpolation approximations (lower order) to speed up rendering.
- Caches are loaded at the start of a lighting session along with simplified geometry for shadow queries.
- Light shaders are individually executed on a graphics processor (GPU) using the data contained in the image-space caches.

A light is re-executed only when its parameters change, permitting scenes with hundreds of lights to be re-rendered at interactive rates.

- Lighting controls are identical to those provided in our primary modeling and animation system, thus allowing the use of complex user-interface paradigms, such as dragging highlights, reflections, and shadows to desired locations. Also, real-time playback of lit shots is provided with low latency, which facilitates iterative refinement.

1.2 Summary of contributions

The lpics system was first deployed in the summer of 2003 and has been used in the production of two animated feature films. It has dramatically improved productivity in master lighting and character light rigging. On typical production shots, lpics reduces render times by *four to six orders of magnitude*, from over 2 hours to 10 frames per second at video resolution (720x301). Detailed results are presented in Section 4. In some cases the quality of feedback is high enough to permit final shot lighting without resorting to conventional software rendering. A comparison of the images generated by lpics and a conventional RenderMan renderer is presented in Figure 1, demonstrating only a minor loss of accuracy.

1.3 Related work

Most relighting engines are based on the assumption that changes in camera movement and object transformations happen less frequently than shading changes, allowing the renderer to cache visibility evaluation in data structures often referred to as *deep framebuffers*. One example of these approaches is the G-buffer [Saito and Takahashi 1990], a collection of images containing the sampled geometric data required for rendering; a similar approach is used in Pixar's lighting tool, Irma [Pixar 2004]. This basic idea was extended considerably by Sequin and Smyrl in the context of ray-traced scenes, where a full ray tree is cached for each pixel and subsequently re-evaluated for surface or light shader changes [Sequin and Smyrl 1989]. Briere and Poulin extended these ideas further by introducing visibility acceleration structures that allow for small changes in object position [Briere and Poulin 1996].

Recent improvements in the performance of graphics processors (GPUs) and hardware shading [Peercy et al. 2000; Proudfoot et al. 2001; Mark et al. 2003] have revived interest in relighting engines. GPU shading alone is generally insufficient for interactive lighting tasks in scenes with complex geometry and expensive surface shaders. However, deep framebuffers provide a compact representation of partially shaded geometry that can be transferred quickly to a GPU for final shading, essentially using the GPU as a generalized stream processor rather than a geometry processor. An example of this approach for simple light shaders is reported in Gershbein and Hanrahan [Gershbein and Hanrahan 2000] based on ideas previously presented in [Lastra et al. 1995]. Our approach, and the one independently-developed by Ragan-Kelley [Ragan-Kelley 2004], also follows the same principles but for more complex environments: a cache-generating shader is executed by a conventional software renderer, followed by a cache-reading shader executed on the GPU. In our system light shaders are manually translated to a hardware shading language, and the results computed by lights are combined by a fixed general-purpose illumination model. At the price of scalability, Ragan-Kelley's system automatically slices the original shader into pre-lighting and post-lighting stages, permitting arbitrary illumination models, and automatically translates (most) shading code to a hardware shading language. These and other differences are discussed in more detail in Section 5.

A final class of relighting algorithms express lighting as a set of basis functions and allow recomputation by performing integration in the space of these basis functions [Sloan 2002; Ng et al. 2004]. While these techniques show promise, they remain limited in their support for arbitrary appearances, and they require pre-computation times that can be overwhelming for typical production scenes.

2 System requirements and design goals

Interactivity yields tremendous improvements in artistic productivity in all stages of the lighting design process. Most lighting tasks begin with *blocking*, in which lights are initially placed and parameters are coarsely adjusted to achieve the desired appearance. In this exploratory phase, small approximations to the final image can be tolerated in exchange for the convenience of interactivity. Blocking is followed by *refinement*, which is characterized by small changes in the parameters of each light (often just the intensity and position). Feedback during refinement must be highly accurate so that decisions can be finalized.

The primary goal of the lpics system is to provide lighting designers with an interactive tool of adequate fidelity for most blocking and many refinement tasks. Interactivity is our highest priority, and in this regard, our system is considerably different from other approaches based on the reevaluation of shaders [Pixar 2004; Ragan-Kelley 2004], which can guarantee image accuracy but may not do so at interactive rates for very complex scenes and shaders (see Section 5).

We found that in order to achieve interactive framerates, carefully chosen approximations must be made. These approximations are acceptable for blocking tasks, such as the initial placement and configuration of lights. In this domain, essential features are correct light shaping and falloff, good approximation of surface response to lights, and accurate positioning of shadows and reflections; on the other hand, features such as accurate antialiasing, motion blur, correctly blurred shadows and reflections are found to be useful but not necessary.

Another requirement for the lighting engine is its integration with the other parts of our production pipeline. Light shader parameters must be identical to permit integration with existing tools, allowing the interactive renderer to offer the same user controls as our standard modeling and animation system. Furthermore the lighting model must be as arbitrary as possible to permit customization by production artists.

A secondary goal of our work was to improve artistic workflow by providing a richer user interface for lighting designers. Fast framerates enable novel interactions, such as the ability to drag highlights, reflections and shadows as in [Pellacini et al. 2002; Gleicher and Witkin 1992], and we expect further innovation in this area in the future.

The major challenge in providing interactive feedback during lighting is the sheer complexity of scenes in our current productions, which includes geometric, surface shader and light shader complexity. The following sections discuss these issues in detail.

2.1 Geometric complexity

Geometric complexity has grown dramatically in recent years, and this trend is expected to continue, while image resolution will grow at a much slower pace. This indicates that algorithms that are bound by image size are probably a better long-term choice than those that depend on scene size. In typical production scenes, geometric complexity arises from the number of objects visible in a given shot as well as from the use of high quality surfaces to represent each

	Figure 1 Scene	Complex Scene
Higher-Order Primitives	2,152	136,478
Shaded points	5,320,714	13,732,520
Surface shaders	152	1,312
Maximum shader length	56,601	180,497
Average shader length	17,268	16,753
Average plugin calls in shader	374	316
Total size of textures used	0.243 GB	3.22 GB
Number of lights	54	169
RenderMan render time	1:30:37	4:09:28

Table 1: Measurements of scene complexity for the scene shown in Figure 1 and a representative high-complexity scene from one of our recent films. Resolution-dependent data is for a 720x301 render.

of these objects. Main characters contain thousands of surfaces, and some include millions of curves representing hair or fur (see Table 1). Indoor environments commonly have tens of thousands of visible objects modeled using subdivision surfaces and NURBS (often in excessive detail), and outdoor environments usually have even higher complexity since they commonly feature millions of leaves or blades of grass. Depth complexity is another aspect of geometric complexity that poses a considerable challenge to relighting engines. Anywhere between 20 and 1000 depth samples are common in final rendering; translucent hair increases depth complexity considerably.

2.2 Surface shading complexity

Shader execution dominates render time in shots without raytracing. This high computational cost is due to several factors. First, separate surface shaders are usually written for each object in a scene, resulting in hundreds or even thousands of unique shaders. These shaders can contain over 100,000 instructions and access several gigabytes of textures in a single scene. Table 1 shows properties of surface shaders in two representative frames from our studio.

Although there are many unique surface shaders, they all tend to have the same structure, based on overlaying a series of basic layers of illumination, each of which is composed of a pattern generation part and an illumination computation. Examples of this use of layered illumination are materials such as rust on paint, dirt on wood, etc. Figure 2 shows pseudo-code illustrating the typical organization of shading code. Pattern generation is the most resource-intensive part of surface shader execution, while the evaluation of illumination is dominated by the cost of light shaders. The surface response to lighting is encoded in a function queried by the light shader at a given point based on parameters computed during pattern generation. We will use the term *illumination model* to describe this function, which is similar in spirit to the BRDF. However, while the BRDF is a four-dimensional function, our illumination model is higher dimensional and can vary based on the number of parameters passed between the light and surface shaders.

RenderMan surface and light shaders permit the expression of arbitrary illumination models. However, our productions typically use only one or two very general ones, just as many different materials can be described by a handful of analytic BRDF models. Our illumination models require positions and normals as well as analogues of diffuse, specular and reflection color, along with numerous additional parameters that modify the surface response. These parameters are only roughly analogous to their physical BRDF equivalents; for example, the component responsible for diffuse-like behavior is view dependent in our model.

```
surface RendermanLayeredSurface(...) {
    for all layers l {
        illumParams[l] = GeneratePatterns(l,...);
        Ci += ComputeIllumination(illumParams[l]);
    }
}

surface LpicsApproximation(...) {
    // Preprocess begins here
    for all layers l {
        illumParams[l] = GeneratePatterns(l,...);
    }
    combinedIllumParams =
        CombineLayerParameters(illumParams);
    // Preprocess ends here

    // This is computed in hardware
    Ci = ComputeIllumination(combinedIllumParams);
}
```

Figure 2: Pseudocode for a layered surface shader and its approximation in the lpics system.

To cope with surface shader complexity, it is essential that a relighting tool caches the results of pattern generation, which are independent of lights. Furthermore, it must be able to quickly reevaluate the illumination as lighting parameters are interactively refined. The adoption of a single illumination model greatly simplifies the matter, as discussed in Section 3.2. The complexity of light shaders is a major obstacle, however.

2.3 Light shading complexity

Lighting complexity results from the number of lights, which is routinely in the hundreds in our production shots, as well as the complexity of the light shaders, which typically contain over 10,000 high level instructions. Our lighting model is an extended version of the one presented in [Barzel 1997]; this lighting model is mostly a direct illumination model with many controls including light intensity, color and shadow. These controls are arranged in different *components* that can be enabled or disabled on each light; examples of such components are shadows, shape and color specifications, falloff, and a variety of other effects. Shadows are a major contributor to light shading complexity, and can be implemented either by shadow mapping, deep shadow mapping [Lokovic and Veach 2000], or raytraced soft shadowing.

While our lighting model is primarily based on direct illumination, certain non-local effects are supported using either environment mapping or raytracing, including reflections, ambient occlusion, raytraced shadows and irradiance. An approximated global illumination solution similar to [Tabellion and Lamorlette 2004] is also available but used infrequently due to its high computational complexity.

Rapid evaluation of light shaders is an essential requirement of a relighting engine. The techniques we employ to address these issues are discussed in Section 3.2.

3 System description

To summarize the previous sections, the main design goal of the lpics system is to re-render at interactive framerates with minimal image approximations in scenes with very high geometric and shading complexity. To cope with geometric complexity, a batch render

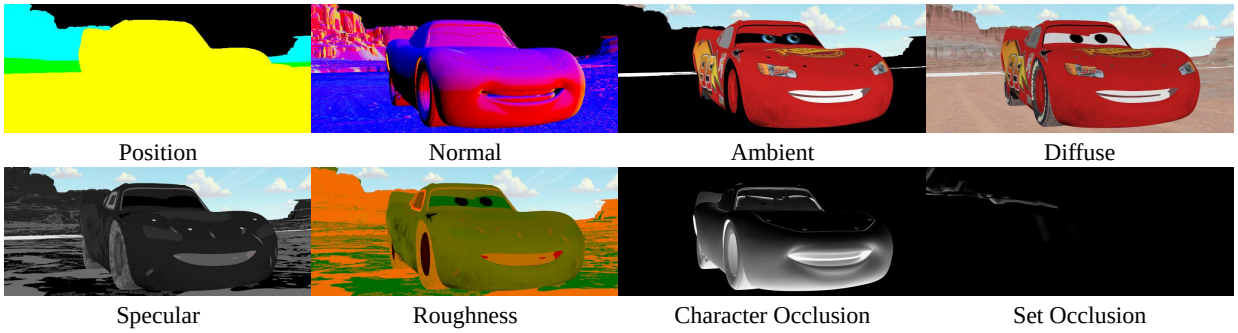


Figure 3: Lpics caches corresponding to the scene in Figure 1

caches per-pixel positions and normals in deep framebuffers. Surface shader complexity is addressed by adopting an illumination model with parameters that are numerically estimated from data cached by the original surface shaders in other deep framebuffers. Finally lighting complexity is dealt with by applying carefully chosen simplifications and translating light shaders to a hardware shading language for execution on a GPU. Light shaders are also specialized to eliminate unnecessary and redundant calculations. The rest of this section describes these techniques in further detail.

3.1 Lpics caches

The first step of our approach is to instrument surface shaders to cache data that will subsequently be read by light shaders. These caches, which we call "lighting pictures" or *lpics*, are essentially deep framebuffers that record all the parameters that contribute to our illumination model. For example, the lpics caches for the scene in Figure 1 are shown in Figure 3.

This step is relatively straightforward because our productions use only a handful of illumination models that are implemented by code in shared libraries. We rely on the fact that these illumination models employ a linear combination of their parameters (e.g. the diffuse, specular, and reflection components) using sums and over operators. This allows us to accurately estimate combined illumination after executing light shaders.

Our interactive render is mostly agnostic about the contents of the lpics caches, requiring only position information for raytraced shadow evaluation. This allows our system to serve multiple productions simultaneously despite their use of different cached data. More importantly, this allows production artists to change the illumination model and lighting model arbitrarily. For example, raytraced reflections, new illumination models, more layers of illumination, and additional layers of transparency can all be handled by simply adding lpics layers.

A drawback of deep framebuffer approaches is that they trade shading throughput for higher latency in camera movement, a tradeoff often acceptable for lighting purposes, in which the camera position is not adjusted interactively. Our system supports multiple camera views by rendering different caches, a technique that is also used to provide high-quality animation previews.

Support for transparency can be provided by multiple sets of lpics caches that are generated using a technique similar to depth peeling [Everitt 2001]. However, our currently deployed system does not support arbitrary transparency because the increased shading complexity would limit interactivity. Scenes of very high depth complexity pose problems because a lack of samples leads to a poor estimation of the response of numerous surfaces. For such scenes, hair shells and other stand-ins are used when generating the lpics caches, thus reducing the fidelity of the computed images.

	Simple	Average	Heavy	Upper bound
Components	8	15	42	58
Instructions	1045	1558	3661	5220
Registers	12	16	18	20
Textures	3	5	13	22

Table 2: Properties of typical lpics GPU light shaders, after specialization. The corresponding non-specialized RenderMan shader contains about 10,000 high-level instructions. Textures exclude input lpics caches.

3.2 Interactive lighting engine

The lpics relighting engine loads lpics caches and transfers them as needed to the GPU as textures. The contribution of each light is then calculated in turn by drawing a screen-aligned quad shaded using a light shader whose parameters are the lpics caches, along with user-specified light parameters. The result of each light is cached and updated only if its parameters are interactively modified, allowing scenes with hundreds of lights to be rendered interactively. Once the caches for each light have been computed, the lighting engine accumulates the results into a final picture using the GPU.[1]

We manually translated our production light shaders to Cg [Mark et al. 2003] for execution on the GPU. The Cg light shaders have the same structure as their RenderMan counterparts, except that they also include the evaluation of the illumination model. The Cg shaders also incorporate simplifications that were carefully chosen to balance speed and image quality. Common areas of simplification were simplified antialiasing, filtering, and interpolation primitives. While this approach might seem time consuming, our productions typically use only a handful of custom light shaders. We briefly investigated techniques for automated translation of RenderMan shaders to Cg but quickly realized its futility in the absence of automated shader simplification.

In fact, even hand-translated shading code cannot run as-is at interactive rates on current GPUs due to the complexity of our lighting model. As illustrated in Table 2, even fairly simple lights have a large number of instructions, registers, and other resources.

Since our lighting code allows lights to have arbitrary numbers of components that control shaping, falloff, shadows, etc., a simple

[1]This step relies on the fact that our illumination model employs a linear combination of lights, except for environment maps, which must be Z-composited.

kind of program specialization [Jones et al. 1993] proved essential. Each light shader is constructed with a macro preprocessor to contain only the code necessary to compute the components present in that light. This is an especially important optimization on graphics hardware where true branching is not supported and function calls are always inlined. For hardware that supports branching instructions, this optimization continues to be useful due to branching overhead and limits on uniform input parameters. Additionally, we have experimented with the program specialization capabilities in the Cg compiler and realized 10-15% speed improvements, which was too small to justify deployment.

In some cases these shaders cannot be executed in a single pass due to resource limitations; in these cases, our system disables light components that are deemed to have low priority. While resource virtualization was impractical at the time the system was deployed, advances in hardware architectures and new algorithms for resource virtualization [Riffel et al. 2004] will soon address this issue.

3.3 Interactive shadows

When computing the lighting at each sample, shadow information might be needed. Our system supports shadow queries using either shadow mapping or raytracing. For shadow mapping the interactive renderer simply uses the same shadow setup as a conventional RenderMan renderer, but it rasterizes geometry using the GPU. Interactivity is achieved by using a coarse tessellation for the geometry as well as implementing various culling optimizations. In our experience, this solution works well for our productions. The only effect that could not be achieved interactively was accurate percentage closer filtering [Reeves et al. 1987], which in our scenes often requires high sampling rates (roughly 100 samples per query); the solution was simply to use built-in hardware shadow mapping, thus falling back on hard shadowing. Improved conditional execution in GPUs will allow us to support this feature in the near future.

For raytraced shadows, our system integrates an interactive raytracer that for each lpics layer generates an image of the result of the shadow query. The results are then bound to the light shaders as an input texture. Due to its cost, raytraced shadowing is implemented as an asynchronous thread in a manner similar to [Tole et al. 2002]. From the user's perspective, raytraced shadows briefly disappear while the user repositions a light, and progressively reappear when the new light position is established.

Our system allows shadows, reflections, and highlights to be directly manipulated, in a manner similar to [Pellacini et al. 2002; Gleicher and Witkin 1992], rather than being indirectly manipulated via lighting parameters. This has proved invaluable to lighting designers, since art direction often imposes precise demands on the placement of highlights and reflections.

Other raytrace-based effects, such as reflections, ambient occlusion and irradiance, are not computed interactively; their results are pregenerated and cached in screen space using the offline renderer and then read in as textures in the hardware shaders. These effects are so expensive to compute that this process mimics the workflow that artists employ even when using the offline renderer. Raytraced reflections can also be supported in our system by caching the illumination model parameters of each ray query in a new lpics layer.

4 Results

Figure 1 and Figure 4 compare the images rendered interactively by lpics with those generated by a conventional batch renderer, RenderMan [Pix 2000]. These scenes are typical of our production shots, with complex geometry, detailed surface shading and lighting. The images reveal slight differences that are typical of images rendered by lpics; in particular sampling and filtering differences are common, since RenderMan textures and shadows are filtered with far more samples than lpics.

We typically run lpics at a resolution of 720x301 on a dual 3.4 GHz Xeon with 4 GB of RAM and a Quadro FX 4400 graphics card. Framerates for production scenes range from 3 to 60 frames per second when interactively manipulating a single light. Performance varies based on the complexity of the light shaders and shadow queries. Final render times for these scenes at the same resolution take from a few hours to more than ten hours when raytracing is heavily used. Excluding shadow queries, our renderer spends most of its time executing light shaders. This holds promise for even higher performance as graphics hardware architectures mature. We have benefited already from rapid advances in GPU performance in the transition from the NV30 to NV40 class of NVIDIA hardware; our original system was deployed to production on NV30 class hardware; when NV40 hardware became available, lpics performance improved by a factor of ten, allowing us to render with more complex lights with no change to our system architecture.

The interactive renders employ a single layer of lpics caches containing 12 parameters including position, normal, diffuse, specular and reflection coefficients, totaling 22 floating-point values. The values are packed into eight tiles of a single texture to work around a limit on texture inputs in current hardware shaders. Cache generation requires about the same amount of time as a few conventional renders, making it cheap compared to some data-driven relighting schemes [Ng et al. 2003].

In practice lpics is also used non-interactively to render preview sequences. While a single frame often requires a tenth of a second to re-render interactively when one light is being manipulated, a full render of the contributions of all lights with newly loaded lpics is often much slower. Nevertheless, a sequence of frames that takes an entire day to complete on our render farm is typically rendered by lpics in a few minutes, which can deliver tremendous savings in time and machine resources.

5 Discussion

Slicing is an alternative approach to the manual instrumentation required for lpics cache generation. General program slicing is a well established technique in the compiler community [Horwitz et al. 1990], and its application to shaders has been explored by [Guenter et al. 1995], where it is called *data specialization*. Slicing would automatically cache the pattern generation portion of surface shaders, which are the most resource intensive parts, and relighting would require only running the residual portion of each shader, consisting of an illumination loop that executes the lights and combines their results. This technique is the essence of recent work by Ragan-Kelley, whose system is in other respects very similar to our own [Ragan-Kelley 2004]. Unfortunately, due to the high number of shaders in our scenes, slicing each surface shader would result in a large number of residual shaders, making it difficult to amortize the cost of binding them to their arguments in the interactive renderer. Furthermore, due to the high number of possible illumination layers, the memory requirements would be too high to be feasible in our scenes. The automation of caching is appealing, although in practice we find the overhead of manual instrumentation to be small. Automated slicing also depends on program dependency analysis, which can be overly conservative, and would fail to handle shader plugins written in arbitrary C code, which are widely used in our productions.

Another aspect of Ragan-Kelley's work that is appealing is automatic translation of light shaders (and residual portions of surface shaders) into hardware shaders. We have explored this topic in

a) lpics render b) final render

Figure 4: Images rendered using our interactive lighting system (a) compared to the final renderers using RenderMan (b).

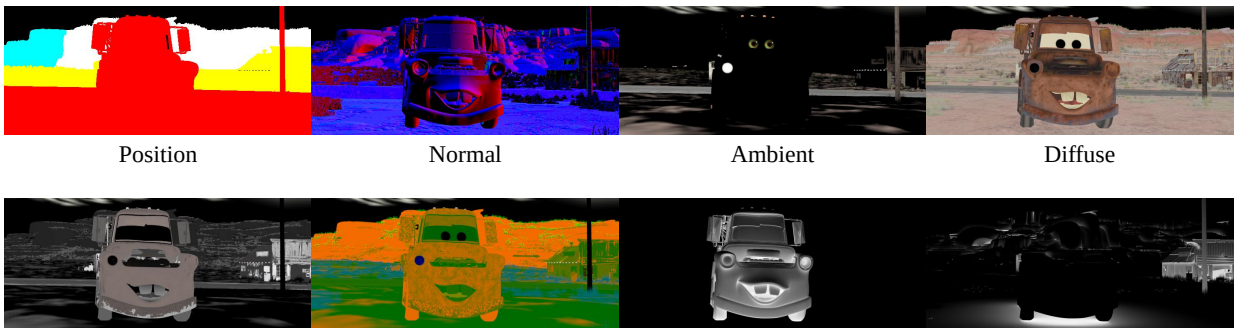


Figure 5: Lpics caches corresponding to the scene in Figure 4

some depth. The first issue we encountered in trying to perform this automatic translation was the inability to map all capabilities of the software renderer onto the currently available graphics hardware. Examples of these features are derivatives and shader plug-ins. Derivatives are available on some graphics hardware, but they are highly discontinuous and computed for each 2x2 block of pixels. RenderMan on the other hand computes continuous derivatives across a surface at the granularity of micropolygon vertices [Cook et al. 1987]. These and other features (such as asynchronous message passing) made automatic translation impractical for our productions. Ragan-Kelley takes the approach of translating only a subset of the RenderMan shading language; while we believe that automatic translation might be superior solution in the future, current approaches are not ready for production use.

An alternative approach to dealing with surface shader complexity in a relighting engine would be to try to represent surface shading response in a purely data-driven fashion by sampling the scene multiple times for each degree of freedom of the light that can affect the illumination similarly to [Ng et al. 2003]; this approach scales well with shading complexity since it turns all surface responses in a common form that can be manipulated independently of the original shaders, thus becoming independent on the number of surface shaders. The main drawback of this data-driven simplification is that, in our lighting environment, lights can affect surface response with between 3 to 75 degrees of freedoms. Nonetheless, we tested this possibility for just incoming direction as in [Ng et al. 2003], thus turning our surface shaders into BRDF-like responses, unfortunately with very poor results. The major issues seemed related to the fact that our surface appearances are very NPR in nature and were not captured well using a BRDF-like representation; adding even more degrees of freedom just made the sampling stage impractical.

Of these two approaches, shader slicing guarantees correctness by re-evaluating the illumination models procedurally, but lacks guarantees in interactivity. On the other hand, data-driven simplification scales well with complexity since it uses a common representation that ignores the original surface shaders, but lacks guarantees in quality (and did not perform well in our tests). In order to get the best of the two worlds, we devised a scheme where we compute only one general illumination model for each pixel whose parameters are estimated numerically to combine all original illumination layers. The illumination model we use is general enough to capture almost every aspect of illumination used in our studio, and the approximations introduced by compressing multiple layers into one are very minor.

6 Conclusions and future work

In this paper we presented a hardware accelerated relighting engine that guarantees interactive feedback while lighting complex environments for computer-animated feature films. The main design goal of our system is to guarantee framerate with minor loss of image fidelity. This goal made our system capable of supporting blocking tasks and allowed the integration of new user interface paradigms. While the original design was geared towards fast blocking, the image fidelity of our system was high enough to support many tasks in final lighting; in fact certain lpics-rendered shots have been approved as final with little verfication using a software render.

Our system scales well with geometric, surface shading and light shading complexity by using a deep framebuffer approach in which samples are numerically estimated and shaded on the GPU. In order to have strong framerate guarantees, our system incurs a small decrease in image fidelity which is noticeable in particular in hair geometry, soft shadowing and very rarely in surface shaders. Even under these conditions, it is our belief that the framerate guarantees

of our system completely justify the small approximations incurred, as proven by the adoption rate of the system in our studio, where it has been used in the lighting of almost every shot since its introduction.

The results demonstrated by our system are very promising, showing that real-time lighting of complex computer generated imagery is possible. We believe that the most fruitful area of improvement is to reduce the fidelity gap between real-time and offline algorithms, by introducing solutions for approximate indirect illumination, soft shadowing, and hair-like geometry. We would also like to automate the treatment of shading simplifications, by developing more robust automatic translation schemes as well as robust shader level-of-details approaches.

Beyond cinematic relighting, we believe that our techniques are broadly applicable to the rendering of high-complexity environments. In particular, we think that the need to approximate shaders will become more important as shader authoring environments encourage artists to create larger shaders. Our work shows the need for two different kinds of approximation. While in the case of surface shading, our system extracts illumination parameters thus converting the surface response to a simpler illumination model numerically, our light shaders are approximated procedurally to maintain a higher quality. We expect other shading systems using high-complexity shaders to follow our steps in introducing (possibly automatically) numerical and procedural approximations to achieve an appropriate speed-vs-quality tradeoff. We also expect our work to foster research into new user interfaces for lighting in highly complex environments, an area that has not yet been explored for lack of an appropriate rendering solution.

7 Acknowledgements

We would like to thank Adam Finkelstein, Dan McCoy, Mark Meyer, Brian Smits, Eliot Smyrl, and Mark VandeWettering from Pixar for their contributions to the shader, raytracer, and direct manipulation widget implementations. Craig Kolb and Nick Triantos from NVIDIA provided valuable Cg and video driver support.

References

BARZEL, R. 1997. Lighting controls for computer cinematography. *Journal of Graphics Tools 2*, 1, 1–20.

BRIERE, N., AND POULIN, P. 1996. Hierarchical view-dependent structures for interactive scene manipulation,. In *Computer Graphics Annual Conference Series 1996*, 89–90.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, vol. 21, 95–102.

EVERITT, C., 2001. Interactive order-independent transparency. NVIDIA White Paper.

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.

GLEICHER, M., AND WITKIN, A. 1992. Through-the-lens camera control. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, vol. 26, 331–340.

GUENTER, B., KNOBLOCK, T. B., AND RUF, E. 1995. Specializing shaders. In *Proceedings of ACM SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, 343–349.

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

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*.

LASTRA, A., MOLNAR, S., OLANO, M., AND WANG, Y. 1995. Real-time programmable shading. In *1995 Symposium on Interactive 3D Graphics*, 59–66.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 385–392.

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.

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.

NG, R., RAMAMOORTHI, R., AND HANRAHAN, P. 2004. Triple product wavelet integrals for all-frequency relighting. *ACM Transactions on Graphics 23*, 3 (Aug.), 477–487.

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., TOLE, P., AND GREENBERG, D. P. 2002. A user interface for interactive cinematic shadow design. *ACM Transactions on Graphics 21*, 3 (July), 563–566.

PIXAR. 2000. *The Renderman Interface*.

PIXAR. 2004. *Irma Documentation*.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, 159–170.

RAGAN-KELLEY, J. M., 2004. Practical interactive lighting design for RenderMan scenes. Undergraduate thesis, Stanford University.

REEVES, W. T., SALESIN, D. H., AND COOK, R. L. 1987. Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, vol. 21, 283–291.

RIFFEL, A. T., LEFOHN, A. E., VIDIMCE, K., LEONE, M., AND OWENS, J. D. 2004. Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Graphics Hardware*, 35–44.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-D shapes. In *ACM SIGGRAPH Computer Graphics*, 197–206.

SEQUIN, C. H., AND SMYRL, E. K. 1989. Parameterized ray tracing. In *Computer Graphics Annual Conference Series 1989*, 307–314.

SLOAN, P.-P. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting, environments. *ACM Transactions on Graphics 21*, 3 (July), 477–487.

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

TOLE, P., PELLACINI, F., WALTER, B., AND GREENBERG, D. P. 2002. Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics 21*, 3 (July), 537–546.