A Fast Relighting Engine for Interactive Cinematic Lighting Design

Reid Gershbein Pat Hanrahan
Stanford University

Abstract

We present new techniques for interactive cinematic lighting design of complex scenes that use procedural shaders. Deep-framebuffers are used to store the geometric and optical information of the visible surfaces of an image. The geometric information is represented as collections of oriented points, and the optical information is represented as bi-directional reflection distribution functions, or BRDFs. The BRDFs are generated by procedurally defined surface texturing functions that spatially vary the surfaces' appearances.

The deep-framebuffer information is rendered using a multi-pass algorithm built on the OpenGL graphics pipeline. In order to handle both physically-correct as well as non-realistic reflection models used in the film industry, we factor the BRDF into independent components that map onto both the lighting and texturing units of the graphics hardware. A similar factorization is used to control the lighting distribution. Using these techniques, lighting calculations can be evaluated 2500 times faster than previous methods. This allows lighting changes to be rendered at rates of 20Hz in static environments that contain millions of objects with dozens of unique procedurally defined surface properties and scores of lights.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation;

Keywords: Animation, Illumination, Image-based Rendering, Optics, Rendering Hardware, Rendering, Texture Mapping

1 Introduction

Cinematography and lighting are extremely important aspects of movie making, as shown by the visual richness of the computer animated movie *Toy Story 2* and the special effects in *Star Wars: The Phantom Menace*. The cinematic goals of an animated computer-generated film are the same as a live-action picture: add depth and clarity to the images, create mood and atmosphere, and focus the viewer's attention on the action [5]. The similarities between the real and virtual media do not end there. Both use elaborate sets and perhaps thousands of props and characters; both use visually complex materials and detailed surfaces; and both require scores of light sources.

However, there is one aspect of designing and evaluating lighting in computer-generated films that is different than a live-action film. In a live-action film the lights are moved and positioned by the grips and the visual result can be seen immediately by the cinematographer. Currently, the opposite is true for computer-generated cinematography. After the cinematographer directs the lighting artist to move a light, the scene must be rendered again. This is a major bottleneck because production rendering systems require minutes to hours to render scenes of movie complexity. The slow rendering rate limits the turnaround time and, ultimately, the productivity of the lighting artist. Moreover, since lighting artists are responsible for creating the final visual appearance of the film, lighting is one of the final stages in production and is done under extreme time pressure. Thus, building an interactive cinematic lighting system would greatly benefit the computer animation industry.

The goals of interactive lighting design are easy to state. What a lighting artist wants is the ability to add, remove, and change the position of a light in real-time. It should also be possible to modify any of the light's attributes such as beam width, fall-off or flap position. Hard and soft shadows are also important. Finally, it is important that the quality of the final image be maintained; approximations that reduce quality are undesirable.

It should be noted that it is rare to change the surface properties of the objects and characters in the scene during the lighting phase. Controlling the look by changing the lights is part of the artistic legacy of traditional film making and is emulated by computer animation production houses. Controlling surface properties on a shot by shot basis is largely impractical in the real-world; and even in the world of computers the properties of most characters are set once and not changed as the characters are reused in different shots. Most previous computer lighting design systems allow modification of *both* surface and light source properties and, unlike our system, have not been optimized for only lighting design.

In this paper, we present a simple rendering engine for interactive lighting design. Like previous systems it uses a deep-framebuffer to store the geometric and optical information of the visible surfaces from a fixed viewpoint. One main innovation is to treat this deep-framebuffer as a set of oriented surface samples so that they may be rendered using the OpenGL graphics pipeline. The state of the graphics pipeline is set to evaluate the reflection from the oriented surface sample due to a single light source. The second innovation is that the optical properties of a surface sample are reduced to only the bi-directional reflection distribution function (BRDF). All spatially varying procedural and texture-mapped surface attributes are evaluated during the process of producing the deep-framebuffer. Our experience is that this leads to a significant speedup, since the majority of the time spent in most surface shaders goes into calculating textures and not reflection models. The final innovation is to enhance the builtin OpenGL BRDF and light source models to be more like those used in production quality rendering systems. Inspired by techniques recently developed by Heidrich and Seidel[14], we factor the BRDF and light source into independent components that may be evaluated using multiple rendering passes that use both the lighting and texturing units. This collection of simple techniques increase the rate at which lighting calculations are computed by at least three orders of magnitude and allow the system to render 512 by 256 images of arbitrarily complex environments with procedural surface and lighting functions at approximately 20 Hz.

2 Related Work

Good examples of current lighting simulation systems are those available for buildings and theatrical sets. Lightscape [1] and Radiance [27] are among the best of the systems designed to accurately predict the lighting in architectural spaces. They allow the designer to select a light fixture from a catalog, and they use the outgoing spectral distribution of the light source during the simulation. They also model the effects of skylighting and have the capability to simulate indirect lighting. These systems focus on physical simulation, and do not support programmable shading languages. Dorsey *et al.* have built a system for designing opera lighting [9] [8]. A major feature of this system is its ability to choreograph the complex time-dependent changes in lighting throughout the production.

Barzel discusses the controls and features of the light source model used at Pixar Animation Studios [2]. This model emphasizes the computer cinematographer's need for precise control over the shape and pattern of the light and its interaction with an object. Our system is designed to control light sources such as these.

There have been a number of methods proposed for quickly rendering changes in lighting based on using *linearity* and *basis images*. Linearity implies that the image formed from two lights is the sum of the images formed from each light separately. Linearity also implies that the effect of a change in color or intensity of a light can be efficiently recomputed by scaling the basis image [8].

Interactive methods have been developed for controlling light positions and parameters using inverse methods. Poulin and Fournier present a technique for determining light positions based on the position and orientation of a highlight or shadow boundary [20]. Schoeneman *et al.* describe a system in which the user paints over the image to brighten or darken different regions, and the system solves for the colors and intensities of a set of fixed lights using a constrained least squares approach[22]. In a system presented by Kawai *et al.*, the user specifies desired features of the lighting, such as visual clarity or contrast, and an optimization procedure is used to search for surface properties, such as reflectivities, and light properties, such as colors, intensities, and directions, that maximize the quality of the lighting [15]. All these techniques are important and useful, but in isolation none of them solve the cinematic lighting problem, especially in scenes with more complex shading models and descriptions.

A major method used to accelerate rendering for design purposes is to fix the camera position and create a "deep-framebuffer" image. A deep-framebuffer image contains all the information needed to rerender the scene, assuming only surface properties and light sources are changed. They do this by precomputing all the geometric information and storing it at each pixel. This precomputation makes it unnecessary to repeat geometric calculations such as tessellating the surface, transforming positions, and removing hidden surfaces during rerendering. The oldest technique of this type is orientation coding [3] [25] [13], where a normal vector is stored in the framebuffer and the lighting calculation is tabulated in the colormap. To change the surface or lighting attributes involves only recomputing the colormap entries. The natural extension of this technique is the G-buffer [21]. In the G-buffer, a collection of framebuffers is allocated, one per attribute. The information available in these framebuffers is available for shading calculations. This technique was considerably extended by Sequin and Smyrl [24]; in their system they store a ray tree at each pixel and reevaluate the ray tree to recompute the effects due to changes in surface properties or lights. Perhaps the most sophisticated deep-framebuffer system is the one built by Briere and Poulin [4]. In their system, visibility structures are stored that accelerate the recomputation of shadows and others effects due to reflection and transparency. As a result their system is able to handle a simple moving environment. Unfortunately, the data structures they use require a great deal of memory, and so their system is impractical for very complex scenes.

The lighting system discussed in this paper is designed to be used with procedural shaders [19] [6] [12] [26]. Using procedural shaders significantly complicates the system design for several reasons. First, the lighting system must be general enough to handle arbitrary shading programs. Second, profiles of rendering systems show that 70-90% of the time is spent evaluating procedural shaders. Thus, a simple minded deep-framebuffer approach is only likely to speed up rendering time by 10-30% (the cost of all other calculations).

Guenter *et al.* have described an approach to incrementally evaluate procedural shaders by using partial evaluation and generalized shader specialization[11]. Although this method is very general, it is quite complex. Specialized versions of the shaders must be created for all combinations of surface and light parameters. In additional, partial evaluation requires a complex caching scheme that uses a lot of memory and is relatively difficult to manage. The combination of the methods used in this paper are much simpler, and as a result much easier to adapt to hardware.

Finally, this paper builds on techniques for hardware accelerated point rendering, or splatting [16][10][28]. The advantage of point rendering is that it is a forward mapping technique that runs very efficiently on graphics pipelines. Current widely available commodity graphics cards can render approximately 5 million points per second [7] and in the near future systems may be able to render 50 million or more points per second. The disadvantage of point rendering is that artifacts such as gaps may occur during the rendering process. However, in lighting design the view point is fixed, so no resampling is required. Thus point rendering is a very simple way of building a high performance rendering engine.

3 Lighting Design Process

Sharon Calahan has written an excellent overview chapter on cinematic lighting design in the context of computer graphics [5]. Her chapter describes the goals of lighting and how it contributes to storytelling. She also outlines the process used by professional lighting artists.

There are two scenarios in which a lighting artist works. In the first stage, called *master lighting*, the artist places lights that provide the background lighting for the set and props of the sequence. When working on the master lighting, the artist judges the lighting by rendering the static scene from a few camera positions that are characteristic of those used during the sequence. The second stage, called *shot lighting*, occurs when the artist is working on an animated sequence with a continuous camera position. Shots tend to last between one to ten seconds. In this stage the lighting artists concentrates on lighting the characters and telling the story.

A lighting artist typically uses a key-frame system in which they choose a few key locations in the animation, creates the lighting they desire for these frames, and then renders the entire animated sequence to judge the lighting for the entire shot. This means that the lighting artist spends most of their time designing the lighting for a single image from a fixed viewpoint. Thus, our design decision to optimize our system for a fixed viewpoint matches well existing production practices.

4 Relighting Engine

In this section we describe the basic operation of our lighting design system and our relighting engine. The system has three major stages:

- The first stage converts a shot from a given viewpoint to a deep-framebuffer. During this process the rendering system is used to partially evaluate the shading.

- The second stage is the lighting design stage. This stage takes as input the deep-framebuffer and outputs a set of lights. This system is built upon a hardware-assisted real-time rendering system so that light sources may be interactively moved and their properties changed.
- In the third stage the set of lights are added to the scene graph and the final rendering is performed.

4.1 Deep-Framebuffer

The first step in our system is the generation of the deep-framebuffer. Each pixel in the deep-framebuffer represents a visible surface element and contains all the geometric and optical information needed to perform subsequent lighting calculations.

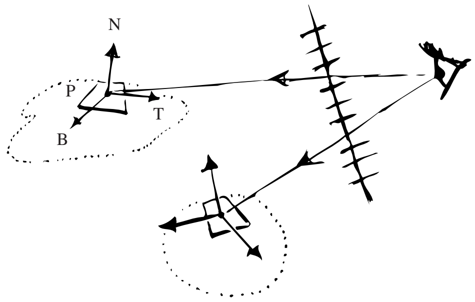

Figure 1: Each sample in the deep-framebuffer stores the world space position, surface normal, tangent, and bitangent vectors of the visible surface.

The geometric information associated with a sample defines the local coordinate system used for shading a point on the surface. This information includes the position, normal, tangent, and bitangent vectors, see Figure 1, all represented in world space coordinates.

For relighting purposes, the only optical information needed is the bi-directional reflection distribution function (BRDF). The BRDF is defined to be the ratio of reflected radiance to the differential incoming irradiance and represents the percentage of light energy reflected in a given outgoing direction for each incoming direction. Figure 2 shows the important direction vectors used in the computation of BRDFs.

In our approach, each BRDF F is represented as a linear combination of primitive BRDFs F_i multiplied by color reflectances C_i

$$F(x, y, w_i, w_o) = \sum_i C_i(x, y) F_i(w_i, w_r, s(x, y))$$

In our system, the primitive BRDFs include the common diffuse and specular (Phong) models, and less common models such as the Cook-Torrance, Oren-Nayar, and a model appropriate for materials such as velvet [17], Specular BRDF models are typically parameterized by a single number, the specularity or roughness, that characterizes the size of the highlight on the surface. Specifically, each deep-framebuffer sample stores an array of n BRDF terms; each term includes a color reflectance, the type of the primitive BRDF, and any associated parameters associated with that primitive BRDF.

Typically, final renderings are done with a production high quality rendering system. Thus, it is necessary to adapt the rendering system to output a deep-framebuffer; this is usually very straightforward. Because we have targetted our system for cinematic lighting, we use rendering systems that have programmable shading languages. To generate the deep-framebuffers, surface shaders that normally output only a final color are modified to output the geometric and optical information described above. In the case of PIXAR's RenderMan shading language [26], this simply requires inserting "print" commands in the right places. However, for the examples used in this paper, we have used a locally written ray tracer as the final rendering system. This ray tracer is similar to RenderMan in that it uses a shading language for creating different surface and lighting models, but the shading language subsystem has been modified to output the required geometric and optical information directly to a file. This system also uses dynamically loaded libraries of C code to evaluate built-in and programmed BRDF models, and this same code may be used by a software-only implementation of the relighting system.

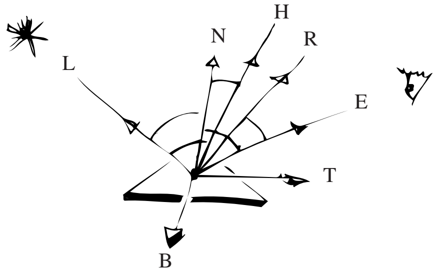

Figure 2: The major vectors required for the evaluation of the BRDF. N, L, and E represent the surface normal, the incoming direction from the light source, and the outgoing direction to the eye, respectively. H is the halfway vector between L and E, and R is the reflection of L about the surface normal N.

The surface sample points and BRDFs are sent down the graphics pipeline to compute lighting. The vertex transformation unit is used to transform the world space location of the point into the shading coordinate system and the lighting unit is used to evaluate the product of the point's BRDF and the radiance from a single light source. Finally, the texturing and compositing features of the fragment processing pipeline are used to modulate the reflected light and blend the result into the framebuffer.

4.2 Reflection Functions

In order to accelerate rendering we factor the color reflectance and BRDF into a form suitable for hardware acceleration.

Deep-framebuffer systems store parameters as images. In our system this image data is converted to texture maps. We store each color reflectance C_i in a texture map in which the location of the color component is the location of its image sample (x, y). When the points are fed into the graphics pipeline, the texture coordinates of each point are set to (x, y) and the color looked up and then used to modulate the color of the lit point.

Heidrich and Seidel recently have shown how a physically-based reflection model may be mapped onto the OpenGL graphics pipeline. The key idea is to factor the reflection model into a product of simpler lower-dimensional terms. Although the general BRDF is a function of 4 dimensions, each term in the factorization is typically a function of only one or two dimensions. For example, the Cook-Torrance reflection model is the product of three terms: a micro-facet distribution function $D(N \cdot H)$, a Fresnel term $F(H \cdot L)$, and a self-shadowing term $S(E \cdot N)S(L \cdot N)$. The micro-facet distribution function and Fresnel term are one-dimensional functions and the self-shading term is a two-dimensional function (actually this may be simplified into a product of one-dimensional functions). The advantage of this factorization is that these lower-dimensional terms may be stored in texture maps.

Building upon their work, we map each reflection function term F_i stored in the deep-framebuffer is mapped to a product of simpler factors. The terms themselves are associated with different stages of the graphics pipeline; BH is a term computed by the host processor, BL by the lighting unit, and BT by texture lookup unit.

$$F_i = (\prod_i BH_i(\theta_i)) \cdot (\prod_j BL_j(\theta_j)) \cdot (\prod_k BT_k(\theta_k))$$

Using this representation, a product of the form $BH \cdot BL \cdot BT$ may be computed using a single rendering pass. If multiple BH, BL or BT factors are required, then multiple passes are used. Note that in general, BH terms are to be avoided if possible, since they do not use the graphics hardware. BL factors are computed using the lighting engine and all that is required is that the normal be passed with each point and the light position be set properly. BT factors are the most complex. To evaluate these terms requires that the texture coordinates of the point be set to the appropriate dot product between two lighting vectors. As a preprocess we must also tabulate the BT factor and store it as a texture map. Note that on newer graphics hardware with multitexturing units, multiple textures may be applied in a single pass and this may reduce the total number of passes.

Our current system uses the following factorization which is automatically created from the relighting engine based on the deep-framebuffer. It is very easy to add additional factorizations or other terms as new reflection models are developed. As future work, it may be possible to compute such a factorization automatically.

$$\begin{aligned} F =& C_D \cdot BL_D(L \cdot N) + C_S \cdot BL_S(N \cdot H, s) + \\ & C_{CT} \cdot BT_{FD}(L \cdot H, N \cdot H) \cdot BT_G(N \cdot L, N \cdot E) + \\ & C_{LN} \cdot BH_{LN}(L \cdot N, k) + C_{EN} \end{aligned} \tag{1}$$

BL_D and BL_S are the standard OpenGL diffuse and specular functions. Following the factorization of the Cook-Torrance reflection model used by Heidrich and Siedel [14], BT_{FD} stores the product of the Fresnel term $F(L \cdot H)$ and the microfacet distribution function $D(N \cdot H)$, and BT_G stores the geometry term $G(N \cdot L, N \cdot E)$. Finally, BH_{LN} and C_{EN} compute reflection components used in the Minneart reflection model [18] (the Minneart model is useful for modeling materials such as velvet [17]). The Minneart model is a sum of two terms $L \cdot N^k$ and $E \cdot N^l$. Since $L \cdot N^k$ cannot be evaluated using the lighting unit, we elected to compute this on the host (although we could have also used a texture map). C_{EN} represents the function $E \cdot N^l$; Because the viewpoint is fixed, $E \cdot N$ is constant for each sample and may be precomputed and stored in a single texture map. Figure 4 shows the the factorization and the resulting passes used for a typical scene. Figure 3 shows the pseudo-code for the multi-pass algorithm.

4.3 Light Sources

Our system is designed to use light sources such as those described by Barzel [2]. The key feature of these light source models is the extreme flexibility in creating lighting distributions. The basic source model specifies the directional beam distribution and the intensity fall-off with distance from the source. Then this spatial lighting distributing is modulated using a set of attenuators. These include procedurally defined superelliptical blockers, “cookie” textures, slide textures, noise textures, and projected shadow mattes.

To implement such lights in real-time we use projective texturing algorithms [23]. To use projective textures in lighting, texture coordinate matrix is set to projectively transform the world space position of the sample into the light coordinate system. The resulting texture map look up is used to specify the color and intensity of the light or the color and alpha of an attenuator. We model a Barzel light as an initial distribution and a product of textures

$$L(x, \omega) = L_s(x, \omega)(\prod_i A_i(\omega))$$

```
/*  Fb = Framebuffer */

Fb = CCT;
BindBTTextures(FD);
Fb = Fb * RenderTextureBRDFPoints(BTFD);
BindBTTextures(G);
Fb = Fb * RenderTextureBRDFPoints(BTG);

BindTexture(CD);
Fb = Fb + RenderHardwareBRDFPoints(BLD);
BindTexture(CS);
Fb = Fb + RenderHardwareBRDFPoints(BLS);

BindTexture(CLN);
Fb = Fb + RenderHostBRDFPoints(BHLN);
BindTexture(CEN);
Fb = Fb + RenderNoBRDFPoints();

BindProjectiveTexture();
Fb = Fb * RenderProjectiveImage();

RenderShadowMap();
Fb = Fb * RenderShadowImage();

If have texture image of previous lights
        Fb = Fb + ImageOfPreviousLights;
```

Figure 3: Pseudo-code for the multi-pass algorithm that corresponds to the pipeline shown in figure 4.

Here L_s is the beam distribution from a point source and A_i are color attenuation maps as a function of direction.

This factorization allows very complex lights to be rendered using multiple passes. However, an important optimization is to premultiply fixed terms together into a single projected texture map so that it may be rendered with a single pass. Another useful mode in the design process is to allow only a single term to be modified. For example, the user changes the shape or size of a single blocker or moves a single attenuator with respect to the light. This mode requires only two passes, one for the static terms and one for the dynamic term.

The computation of shadows requires two additional passes. The algorithm we use takes advantage of hardware z-buffering, depth comparisons, and the SGI OpenGL shadow extension. The first pass of the shadow algorithm creates a shadow depth map from the viewpoint of the light source. In second pass the points are rendered from the eye point, but also transformed into the light source coordinate system. If the depth of the point is less than the corresponding shadow map depth, then the point is in shadow and its alpha value is set to zero, otherwise the alpha value is set to one. Other approaches for computing shadows are available if this functionality is not supported by the graphics hardware, but it is assumed that these features will appear in commodity graphics hardware within the next year.

Finally, we exploit the linearity of the contribution of the individual lights. This property allows us to generate an image of the combined result of the new light source with the previously placed lights by adding the image of the new light to the image of previously computed lighting effects. In our system, this is done by storing the previously computed image as a texture and adding it to the result of the newly computed image in a final pass.

4.4 Performance Optimizations

In order to minimize the total number of primitives processed by the graphics hardware, we cull points that do not contribute to a particular reflection function. This culling is easily performed by checking the color reflectance map: if the color is black (0), then

Rendering	Lights	Time (s)	Speedup
Geometry + Texturing + Lighting	10	121	1:1
Texturing + Lighting	10	51	2.4:1
Lighting (sw)	10	13	9.3:1
Lighting (sw)	1	1.3	93:1
Lighting (hw)	10	0.5	242:1
Lighting (hw)	1	0.05	2420:1
Lighting + Shadows (hw)	1	0.25	484:1

Table 1: Time to render the scene in Figure 4.

that point is culled. Additionally, if the properties of the point are static (this is always the case unless texture coordinates need to be recomputed), we store points in a display list to maximize rendering rate, Finally, we can partition the points by object and only rendering a subset of the points that come from a particular object. This is very useful since it speeds up the system and since lights are often defined object by object.

Another major method used to increase performance is to reorder the calculation to maximize coherence. For example, SGI's Performer and many other scene graph libraries reorder primitives by texture maps; that is, all the primitives that use the same texture map are drawn together. We have found that we can significantly speedup rendering by reordering the points by the material properties. In one sense this is done by reordering the shading calculation so that all diffuse calculations are done in one pass and all specular calculations are done in another pass. As a further optimization, we quantize material parameters such as the specular exponent to reduce the number of different materials and to reorder rendering so that all points with the same properties are rendered together. Since there is substantial cost to switching material properties, this is a significant speedup. In the case of the specular exponent, we found applying a logarithmic transformation before quantization is also useful.

5 Results

We have implemented a simple lighting design system using our rendering engine. A detailed discussion of all the features of this system is beyond the scope of this paper. In this section we describe various experiments we did to estimate the performance of the system and to validate our design decisions.

We created a simple test scene typical of a set (see Figure 4). All the shaders are programmed in a shading language very similar to the RenderMan shading language; this scene contains 104 objects, 2.2 million micro-triangles, and 13 distinct surface shaders. Figure 5 presents timings of various stages. We timed how long it took to render the scene using the ray tracer and how long it took to render the scene using our deep-framebuffer. We also compared the time it took to render the deep-framebuffer using software vs. hardware. All timings were done using a Onyx2 with a 195 MHZ MIPS R10000 and an InfiniteReality2 Graphics Subsystem. The final image size was 512 by 256 which represents 131,072 samples; a detailed breakdown of the number of points required per pass is shown in figure 4. The shadow timing includes the generation of a shadowmap from 100,000 polygons of geometry.

There are seven separate passes in figure 4, not including shadow computations (the two passes of BT_{FD} and BT_G are merged together in the diagram). The average number of points rendered per pass is 48,210.

Some results stand out. First, in this scene approximately 42% of the time is spent in geometry processing and 58% in shading. Second, of the shading, 25% of the time is spent evaluating the reflection model and 75% of the time is spent evaluating spatially varying textures that control the reflection model. Third, as expected, there is linear speedup in the number of lights; that is, in a scene consisting of ten lights, moving one light is exactly ten times faster than rendering all the lights. Finally, the hardware version is 27.5 times faster than the software system. Overall the hardware relighting system runs 2000 times faster than the software batch renderer. Moreover, the lighting design system runs at roughly 20 Hz, fast enough for interactive use.

6 Summary and Discussion

We have described and implemented a simple rendering engine for interactive lighting design. The system meets our original goals which were to accept scene descriptions with programmable shaders from a production rendering system; and to allow a lighting artist to position the light source and modify all the major light properties in real-time without compromising image quality. Figure 5 shows examples of lighting created using our system. Although our example set is relatively simple, we believe our system will scale to movie-complexity scenes since the rendering time only depends on the output complexity of the deep-framebuffer.

The shadow computation is dependent on the input complexity of the scene, while all other computations are based on the output complexity of the deep-framebuffer. Therefore, the shadow algorithm does not scale as well as the other computations with increases in scene complexity. We are currently exploring new image and point-based methods for shadow approximations.

The key to our system is adapting deep-framebuffer technology to modern graphics hardware. Over the past several years we have built a series of more and more complex software-only rendering engines for lighting design with limited success. There is no silver bullet: interactive relighting requires lots of floating point operations and today's graphics hardware delivers more flops per dollar than conventional CPUs. For example, a 1999 single chip graphics accelerator from NVIDIA is rated by the manufacturer at 50 gigaflops and costs tens of dollars. (We are currently in the process of porting our system from the SGI IR used in this paper to a PC.) In the future the gap between graphics hardware and the main processor is predicted to be even larger, and so our approach will be even more attractive. For example, a hardware system capable of rendering 50 million points per second would allow us to do more than 20 passes at 20 Hz on a 512 by 256 image. Finally, graphics hardware vendors are introducing new features such as multitexturing and texture combiners that support advanced shading and even programmablity. With hardware assistance, we believe interactive lighting design will soon be as common as interactive 3d painting.

Acknowledgements

We would like to thank Matt Pharr and Craig Kolb for codeveloping the technique of separating surface texturing from the lighting calculations. Julie Dorsey for discussions on lighting design systems. Pixar's RenderMan Graphics R&D group for years of discussions on production lighting and rendering systems. Kekoa Proudfoot and Bill Mark for their discussions on real-time programmable shading systems and graphics hardware. Sudeep Rangaswamy for creating the chair, table and bookshelf models in our test images. Reid would like to thank Sharon Calahan for her guidance and teaching of cinematography and lighting design during the productions *A Bug's Life* and *Toy Story 2*. This research was supported by NSF contract number CCR-9508579-001, and DARPA contracts DABT63-96-C-0084-P00002 and DABT63-95-C-0085-P00006.

References

[1] Lightscape visualization system.

[2] Ronen Barzel. Lighting controls for computer cinematography. *Journal of Graphics Tools*, 2(1):1–20, 1997.

[3] J. F. Blinn. Raster graphics. In K. Booth, editor, *Tutorial: Computer Graphics*. IEEE Computer Society, 1979.

[4] Normand Briere and Pierre Poulin. Hierarchical view-dependent structures for interactive scene manipulation. In *Computer Graphics Annual Conference Series 1996*, pages 83–90. Siggraph, August 1996.

[5] Sharon Calahan. Storytelling through lighting, a computer perspective. In Tony Apodaca and Larry Gritz, editors, *Advanced Renderman: Creating CGI for Motion Pictures*, pages 337–382. Morgan Kaufman Publishers, San Francisco, 1999.

[6] Robert L. Cook. Shade trees. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 223–231, July 1984.

[7] NVIDIA Corporation. *GeForce 256 Graphics Accelerator Specifications*, December 1999.

[8] J. Dorsey, J. Arvo, and D. Greenberg. Interactive design of complex time-dependent lighting. *IEEE Computer Graphics and Applications*, 15(2):26–36, March 1995.

[9] Julie O'B. Dorsey, Francois X. Sillion, and Donald P. Greenberg. Design and simulation of opera lighting and projection effects. *Computer Graphics*, 25(4):41–50, July 1991.

[10] J.P. Grossman and Willian J. Dally. Point sample rendering. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop*, Eurographics, pages 181–192, 1998.

[11] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. In *Computer Graphics Annual Conference Series 1995*, pages 343–350. Siggraph, August 1995.

[12] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 289–298, August 1990.

[13] Paul S. Heckbert. Techniques for real-time frame buffer animation. In *Computer FX 84 Conference*, October 1984.

[14] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In Alyn Rockwood, editor, *Computer Graphics (SIGGRAPH '99 Proceedings)*, pages 171–178, August 1999.

[15] John K. Kawai, James S. Painter, and Michael F. Cohen. Radioptimization - goal based rendering. In *Computer Graphics Annual Conference Series 1993*, pages 147–154. Siggraph, August 1993.

[16] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical Report 85-022, UNC-Chapel Hill Computer Science, 1985.

[17] Rong Lu, Jan J. Koenderink, and Astrid M. Kappers. Optical properties (bidirectional reflection distribution functions) of velvet. In *Applied Optics*, volume 37, pages 5974–5984, 1998.

[18] M. Minnaert. The reciprocity principle in lunar photometry. In *Astrophys. J.*, volume 93, pages 403–410, 1941.

[19] Ken Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[20] Pierre Poulin and A. Fournier. Lights from highlights and shadows. In *1992 Symposium on Interactive 3D Graphics*, pages 31–38, March 1992.

[21] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Computer Graphics Annual Conference Series 1990*, pages 197–206, August 1990.

[22] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. Painting with light. In *Computer Graphics Annual Conference Series 1993*, pages 143–146. Siggraph, August 1993.

[23] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics Annual Conference Series 1992*, pages 249–252, July 1992.

[24] Carlo H. Sequin and Eliot K. Smyrl. Parameterized ray tracing. In *Computer Graphics Annual Conference Series 1989*, pages 307–314. Siggraph, July 1989.

[25] Kenneth R. Sloan and Christopher M. Brown. Color map techniques. *Computer Graphics and Image Processing*, 10:297–317, 1979.

[26] Steve Upstill. *The RenderMan Companion*. Addison Wesley, 1992.

[27] Gregory J. Ward. The radiance lighting simulation and rendering system.

[28] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, August 1990.

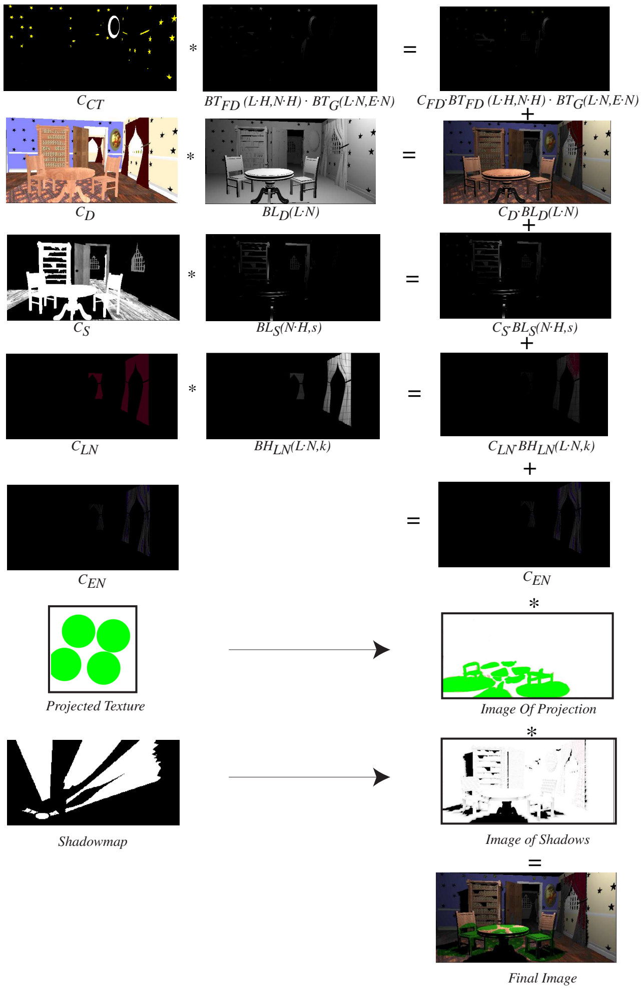

Figure 4: This image shows the stages of the multi-pass rendering algorithm. The number of points rendered per row are: row 1 = 17,302 points, row 2 = 129,604 points, row 3 = 39,006 points, row 4 = 10,246 points, row 5 = 10,246 points, row 6 (projective texture) = 131,072 points, row 7 (shadow map) = 131,072.

Figure 5: This is an example of lighting designed using our system. It took 2 hours to create, contains 60 lights (mostly to simulate inter-reflection effects), and defines an early morning mood.