Advanced RenderMan 2: To RI_INFINITY and Beyond

SIGGRAPH 2000 Course 40

course organizer: Larry Gritz, Pixar Animation Studios

Lecturers:

Tony Apodaca, Pixar Animation Studios
Larry Gritz, Pixar Animation Studios
Tal Lancaster, Walt Disney Feature Animation
Mitch Prater, Pixar Animation Studios
Rob Bredow, Sony Pictures Imageworks

July 2000

Desired Background

This course is for graphics programmers and technical directors. Thorough knowledge of 3D image synthesis, computer graphics illumination models and previous experience with the RenderMan Shading Language is a must. Students must be facile in C. The course is not for those with weak stomachs for examining code.

Suggested Reading Material

The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics, Steve Upstill, Addison-Wesley, 1990, ISBN 0-201-50868-0.

> This is the basic textbook for RenderMan, and should have been read and understood by anyone attending this course. Answers to all of the typical, and many of the extremely advanced, questions about RenderMan are found within its pages. Its failings are that it does not cover RIB, only the C interface, and also that it only covers the 10-year-old RenderMan Interface 3.1, with no mention of all the changes to the standard and to the products over the past several years. Nonetheless, it still contains a wealth of information not found anyplace else.

Advanced RenderMan: Creating CGI for Motion Pictures, Anthony A. Apodaca and Larry Gritz, Morgan-Kaufmann, 2000, ISBN 1-55860-618-1.

> A comprehensive, up-to-date, and advanced treatment of all things RenderMan. This book covers everything from basic RIB syntax to the geometric primitives to advanced topics like shader antialiasing and volumetric effects. For any RenderMan professional, this is the book that will be sitting open on your desk most of the time. Buy it. Love it. Tell your friends.

"The RenderMan Interface Specification, Version 3.2," Pixar, 2000.

> Yes, you read that right – version 3.2. As I write these course notes in March, it's not quite done yet. But we're intending to have it ready in plenty of time for SIGGRAPH. The RI Spec 3.2 is the updated bible of the RenderMan C and RIB API's and the RenderMan Shading Language. It reflects all of the changes and extensions made informally by *PRMan* and *BMRT* over the years, put together in a single, clean document.

Lecturers

Larry Gritz leads the rendering research team at Pixar and is chief architect of the RenderMan Interface Standard. Larry has been at Pixar since 1995, developing new rendering techniques and systems, and periodically serving as a technical director. His film credits include *Toy Story*, *Geri's Game*, *A Bug's Life*, and *Toy Story 2*. Larry's research interests include global illumination, shading languages and systems, and rendering of hideously complex scenes. Prior to joining Pixar, Larry authored the popular *Blue Moon Rendering Tools*, a shareware RenderMan-compliant renderer that supports ray tracing and radiosity. Larry has a BS from Cornell University and MS and PhD degrees from the George Washington University. Larry is co-author of *Advanced RenderMan: Creating CGI for Motion Pictures* (Morgan-Kaufmann, 1999)

Tony Apodaca is a senior technical director at Pixar Animation Studios. In the 20th century, he was director of Graphics R&D at Pixar, was co-creator of the RenderMan Interface Specification, and lead the development of *PhotoRealistic RenderMan*. Tony has been at Pixar since 1986, where his film credits include work from *Red's Dream* through *A Bug's Life*. He received a Scientific and Technical Academy Award from the Academy of Motion Picture Arts and Sciences for work on *PhotoRealistic RenderMan*, and with Larry was co-author of *Advanced RenderMan: Creating CGI for Motion Pictures*. He holds an MEng degree in computer and systems engineering from Rensselaer Polytechnic Institute.

Tal Lancaster has been a technical director for Walt Disney Feature Animation since 1995, concentrating on RenderMan shader development. His film credits include *Fantasia 2000* and *Dinosaurs*. Tal also maintains the "RenderMan Repository": `www.renderman.org`.

Mitch Prater is a shading TD at Pixar, and has been writing shaders for production and product uses for over 10 years. Mitch's film credits include *Toy Story*, *A Bug's Life*, *Toy Story 2*, and numerous commercials and short films. Prior to joining Pixar, he taught computer graphics and worked in graphics and thermal simulation.

Rob Bredow served as senior technical director at Sony Pictures Imageworks, where he just completed work on *Stuart Little*. Prior to his stint at Sony Imageworks, Rob was director of R&D at VisionArt Design and Animation, Inc., where he worked on such projects as *Star Trek DS9*, *Independence Day*, and *Godzilla*. Currently he is working as Visual Effects Supervisor on the film *Megiddo* due to be released early 2001.

Schedule

Welcome and Introduction Larry Gritz	8:30 AM
How *PhotoRealistic RenderMan* Works Tony Apodaca	8:45 AM
Break	10:00 AM
The Secret Life of Lights and Surfaces Larry Gritz	10:15 AM
Lunch	12:00 PM
Rendering Related Issues on the Production of Disney's *Dinosaur* Tal Lancaster	1:30 PM
The Artistry of Shading: Human Skin Mitch Prater	2:30 PM
Break	3:00 PM
The Artistry of Shading: Human Skin (continued)	3:15 PM
Fur in *Stuart Little* Rob Bredow	3:45 PM
Roundtable Discussion / Q&A All panelists	4:45 PM
Finished — go to a party!	5:00 PM

Contents

Preface

Welcome to *Advanced RenderMan 2: To RI_INFINITY and Beyond*. How's that for a goofy title? This course covers the theory and practice of using RenderMan to do the highest quality computer graphics animation production. This is the sixth SIGGRAPH course that we have taught on the use of RenderMan, and it is quite advanced. Students are expected to understand all of the basics of using RenderMan. We assume that you have all read the venerable *RenderMan Companion*, and certainly hope that you have attented our previous SIGGRAPH courses and read the book, *Advanced RenderMan: Creating CGI for Motion Pictures* (Morgan-Kaufmann, 1999).

We've taught several SIGGRAPH courses before. Though this year's course resembles the 1998 and 1999 courses ("Advanced RenderMan: Beyond the Companion") in detail and advanced level, we are delighted to be teaching a course that has all new topics and even some new speakers.

Topics this year include the inner workings of *PRMan*, the operation of lights and surfaces and how you can use that knowledge to create custom lights and materials, advanced aspects of fur and hair rendering, skin and artistic effects, and absurd displacement shading. Not a single topic is repeated from previous SIGGRAPH course. A couple of speakers are veterans, but three are first-time SIGGRAPH RenderMan course speakers!

We always strive to make a course that we would have been happy to have attended ourselves, and so far this year is definitely shaping up to meet or exceed our expectations.

Thanks for coming. We hope you will enjoy the show.

Chapter 1
How PhotoRealistic RenderMan Works

(...and what you can do about it)

Tony Apodaca
Pixar Animation Studios
aaa@pixar.com

Abstract

Beginning (and maybe even intermediate) RenderMan users can use the abstract model of a renderer which is provided by the RenderMan Interface, and described by the *RenderMan Companion* and numerous previous Siggraph courses, to build their models and write their shaders. The whole point of the RenderMan Religion is that this should possible. And the vast majority of the time, this will lead to success.

However, those who are doing the most subtle manipulations of the renderer, making the kind of images that the students in this course would be proud of, need to have a deeper understanding of what is going on inside their chosen renderer. This chapter explains how PhotoRealistic RenderMan works on the inside, and how it sometimes fails to work, as well.[1]

1.1 History

Pixar's *PhotoRealistic RenderMan* renderer is an implementation of a scanline rendering algorithm known as the *Reyes* architecture. Reyes was developed in the mid-1980s by the Computer Graphics Research Group at Lucasfilm (now Pixar) with the specific goal of creating a rendering algorithm that would be applicable to creating special effects for motion pictures. The algorithm was first described by Cook et al. in their 1987 Siggraph paper "The Reyes Image Rendering Architecture".

[1]This section is reprinted from *Advanced RenderMan: Creating CGI for Motion Pictures*, by Anthony A. Apodaca and Larry Gritz, ISBN 1-55860-618-1, published by and copyright ©2000 by Morgan Kaufmann Publishers, all rights reserved, and is used with permission. Hey! Why don't you go get the whole book instead of reading a couple out-of-context excerpts?

They developed this novel rendering algorithm because they felt that the other algorithms generally in use at the time (polygon Z-buffer algorithms, polygon scanline algorithms and ray tracers) had various flaws and constraints that really limited their use in this venue.

Images used in motion picture special effects need to be *photorealistic* — that is, appear of such high quality that the audience would believe they were filmed with a real camera. All of the image artifacts that computer graphics researchers had had to live with up to this point were unacceptable if the images were going to fool the critical eyes of the movie-going masses. In particular, Reyes was designed to overcome the following problems with existing rendering algorithms:

- **Vast visual complexity**: Photographs of the real world contain millions of objects, and every object has minute details that make it look real. CG images must contain the same complexity if they are to blend seamlessly with live camera work.
- **Motion blur**: Photographs of moving objects naturally exhibit a blur due to the camera shutter being open for a period of time while the object moves through the field of view. Decades of stop-motion special effects failed to take this into account, and the audience noticed.
- **Speed and memory limitations**: Motion pictures contain over 100,000 frames and are filmed in a matter of days or weeks. Fast computers are expensive, and there is never enough memory (particularly in retrospect). Implementability on special purpose hardware was a clear necessity.

The resulting design brought together existing work on curved surface primitives and scanline algorithms with revolutionary new work in flexible shading and stochastic antialiasing to create a renderer that could produce images that truly looked like photographs. It was first used in a film titled *Young Sherlock Holmes* in 1985, drew rave reviews for its use in *The Abyss* in 1989, and in 1993 was given an Academy Award for contributions to the film industry.

1.2 Basic Geometric Pipeline

The Reyes algorithm is a geometric pipeline, not entirely unlike those found in modern-day hardware graphics engines. What sets it apart is the specific types of geometric operations that occur in the pipeline, and the way that, as the data streams through the system, it gains and retains enough geometric and appearance fidelity that the final result will have very high image quality. Figure 1.1 shows the basic block diagram of the architecture.

The first step, of course, is loading the scene description from the modeler. Typically, the scene description is in a RIB file, loaded from disk. In that case, the RIB file is read by a RIB parser, which calls the appropriate RI routine for each line of the RIB file. Notice that since the RIB file is a simple metafile of the RI API, it is extremely easy to parse. The only minor complexity arises from the handling of parameterlist data, which is dependent on the parameter type declarations that appear earlier in the RIB file. Alternatively, a program that is linked to the renderer can call the RI API directly, in which case the parser is simply bypassed.

The second step is the processing of the RenderMan Interface calls themselves. This stage of the pipeline maintains the hierarchical graphics state machine. RI calls fall into two classes: attributes or options that manipulate the graphics state machine; and geometric primitives whose attributes are defined by the then-current version of the graphics state machine. The hierarchical graphics state

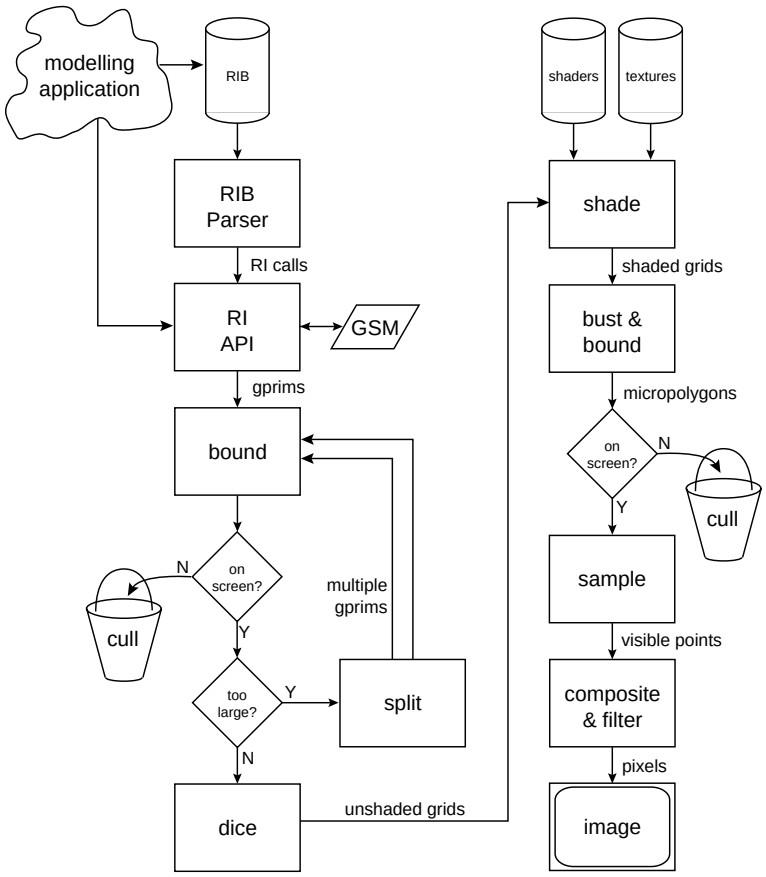


Figure 1.1: The Reyes rendering pipeline.

machine is kept in a stack-based data structure within the RI layer. Whenever a geometric primitive arrives, the current top-of-the-stack set of attributes is attached to the primitive before it proceeds into the main Reyes geometric processing engine.

1.2.1 Splitting Loop

The first thing that Reyes does to arriving primitives is *bound* them. The renderer computes a camera-space axis-aligned bounding box that is guaranteed to contain the entire primitive. RenderMan does not contain any unbounded primitives (such as infinite planes), so this is generally straightforward. Most RenderMan primitives have the convex-hull property, which means that the primitive is entirely contained within the volume outlined by the vertices themselves. Primitives that don't have this property (such as Catmull-Rom patches) are converted to equivalent primitives that do (such as Bezier patches).

Next, the bounding box is checked to see if the primitive is actually on-screen. The camera description in the graphics state gives us the *viewing volume*, a 3D volume of space that contains everything that the camera can see. In the case of perspective projections, this is a rectangular pyramid, bounded on the sides by the perspective projection of the screen window (sometimes called the *screen-space viewport* in graphics texts) and truncated at front and back by the near and far clipping planes; for an orthographic projection this is a simple rectangular box. It is important to note at this point that Reyes does not do any global illumination or global intervisibility calculations of any kind. For this reason, any primitive that is not at least partially within the viewing volume cannot contribute to the image in any way and therefore is immediately culled (trivially rejected). Also, any one-sided primitives that are determined to be entirely back-facing can be culled at this stage, because they also cannot contribute to the image.

If the primitive is (at least partially) on-screen, its size is tested. If it is deemed "too large" on screen, according to a metric described later, it is *split* into smaller primitives. For most parametric primitives, this means cutting the primitive in two (or possibly four) along the central parametric line(s). For primitives that are containers of simpler primitives, such as polyhedra, splitting may mean roughly dividing into two containers each with fewer members. In either case, the idea is to create subprimitives that are simpler and smaller on-screen, and more likely to be "small enough" when they are examined. This technique is often called "divide and conquer."

The resulting subprimitives are then dropped independently into the top of the loop, in no particular order, to be themselves bound, cull-tested, and size-tested. Eventually, the progeny of the original primitive will pass the size test and can move on to the next phase called *dicing*. Dicing converts the small primitive into a common data format called a *grid*. A grid is a tessellation of the primitive into a rectangular array of quadrilateral facets known as *micropolygons* (see Figure 1.2). (Because of the geometry of the grid, each facet is actually a tiny bilinear patch, but we call it a micropolygon nonetheless.) The vertices of these facets are the points that will be shaded later, so the facets themselves must be very small in order for the renderer to create the highly detailed, visually complex shading that we have come to expect. Generally, the facets will be on the order of one pixel in area. All primitives, regardless of original type, are converted into grids that look essentially the same to the remainder of the rendering pipeline. At this stage, each grid retains all of the primitive's attributes, and any primitive vertex variables that were attached to the primitive have been correctly interpolated onto every vertex in the grid, but the vertices have not yet been shaded.

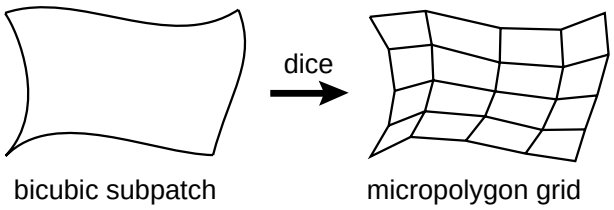


Figure 1.2: A primitive that is small enough will be diced into a grid of tiny bilinear facets known as micropolygons.

1.2.2 Shading

The grid is then passed into the shading system to be shaded. *PRMan* first evaluates the displacement shader, which may move grid vertices and/or recompute shading normals. Then the surface shader is evaluated. In a typical surface shader, there are calls to `diffuse` or `illuminance` somewhere in the code. These routines require the evaluation of all of the light shaders attached to the object. The light shaders run as "coroutines" of the surface shader the first time that they are needed, but their results are cached and reused if they are accessed again (for example, in a subsequent `specular` call). When the surface shader finishes, it has computed the color and opacity of every grid vertex. Finally, the atmosphere shader is evaluated, making adjustments to the vertex color and opacity to simulate fog or other volumetric effects. The end result of all of these evaluations is that the grid now has final color and opacity assigned to each vertex, and the vertices themselves might be in different places than when originally diced. The other attributes of the original primitive are now mostly superfluous.

The details of the method that the shading system uses to assign color and opacity to each grid vertex is of very little consequence to the Reyes pipeline itself but, of course, is of central importance to the users of *PRMan*.

PRMan's shading system is an interpreter for the RenderMan Shading Language, which reads the byte-codes in the `.slo` file previously created by the Shading Language compiler (Hanrahan and Lawson, 1990). Notice that the data structure that the interpreter operates on, a grid, is a bunch of large rectangular arrays of floating-point numbers. For this reason, it may not be surprising that the interpreter actually executes as a virtual SIMD (Single Instruction, Multiple Data) vector math unit. Such vector pipelines were typical in 1980's-vintage supercomputers.

The run-time interpreter reads shader instructions one at a time and executes the operator on all grid vertices. It is important to contrast this scheme with the alternative, which would run the entire shader on the first grid vertex, then run it on the second grid vertex, and so on. The advantages of the breadth-first solution is efficiency. We are able to make near optimal use of the pipelined floating-point units that appear in modern processors, and we have excellent cache performance due to strong locality of reference. In addition, for `uniform` variables, the grid has exactly one data value, not *nvertices* values. As a result, the interpreter is able to compute operations on uniform data exactly once on each grid, saving significantly over the redundant calculations that would be necessary in a depth-first implementation.

However, life is never all gravy. When a conditional or loop instruction is reached, the results may require that not all points on the grid enter the protected block. Because of this, the SIMD controller has *run flags*, which identify which grid vertices are "active" and which are "inactive" for the current instruction. For any instruction where the run-flag vector has at least one bit on, the instruction is executed, but only on those grid vertices that require it. Other operators such as `else`, `break`, and `continue` manipulate the run flags to ensure that the SIMD execution accurately simulates the depth-first execution.

Another advantage of the SIMD execution model is that neighborhood information is available for most grid vertices (excepting those on the grid edges), which means that differentials can be computed. These differentials, the difference between values at adjacent grid vertices, substitute for derivatives in all of the Shading Language operators that require derivative information. Those operators, known generically as *area operators*, include `Du`, `calculatenormal`, `texture`, and their related functions. Notice that grid vertices on edges (actually on two edges, not all four) have no neighbors, so their differential information is estimated. *PRMan* version 3.8 and lower estimated

this poorly, which led to bad grid artifacts in second-derivative calculations. *PRMan* version 3.9 has a better differential estimator that makes many of these artifacts disappear, but it is still possible to confuse it at inflection points.

1.2.3 Texturing

The number and size of texture maps that are typically used in photorealistic scenes are so large that it is impractical to keep more than a small portion of them in memory at one time. For example, a typical frame in a full-screen CGI animation might access texture maps approaching 10 Gb in total size. Fortunately, because this data is not all needed at the highest possible resolution in the same frame, mip-maps can be used to limit this to 200 Mb of texture data actually read. However, even this is more memory than can or needs to be dedicated to such transient data. *PRMan* has a very sophisticated texture caching system that cycles texture data in as necessary, and keeps the total in-core memory devoted to texture to under 10 Mb in all but extreme cases. The proprietary texture file format is organized into 2D tiles of texture data that are strategically stored for fast access by the texture cache, which optimizes both cache hit rates and disk I/O performance.

Shadows are implemented using shadow maps that are sampled with *percentage closer* filtering (Reeves et al., 1987). In this scheme, grid vertices are projected into the view of the shadow-casting light source, using shadow camera viewing information stored in the map. They are determined to be in shadow if they are farther away than the value in the shadow map at the appropriate pixel. In order to antialias this depth comparison, given that averaging depths is a nonsensical operation (because it implies that there is geometry in some halfway place where it didn't actually exist), several depths from the shadow map in neighboring pixels are stochastically sampled, and the shadowing result is the percentage of the tests that succeeded.

1.2.4 Hiding

After shading, the shaded grid is sent to the hidden surface evaluation routine. First, the grid is *busted* into individual micropolygons. Each micropolygon then goes through a miniature version of the main primitive loop. It is bounded, checked for being on-screen, and back-faced culled if appropriate. Next, the bound determines in which pixels this micropolygon might appear. In each such pixel, a stochastic sampling algorithm tests the micropolygon to see if it covers any of the several predetermined point-sample locations of that pixel. For any samples that are covered, the color and opacity of the micropolygon, as well as its depth, are recorded as a *visible point*. Depending on the shading interpolation method chosen for that primitive, the visible-point color may be a Gouraud interpolation of the four micropolygon corner colors, or it may simply be a copy of one of the corners. Each sample location keeps a list of visible points, sorted by depth. Of course, keeping more than just the frontmost element of the list is only necessary if there is transparency involved.

Once all the primitives that cover a pixel have been processed, the visible point lists for each sample can be composited together and the resulting final sample colors and opacities blended together using the reconstruction filter to generate final pixel colors. Because good reconstruction kernels span multiple pixels, the final color of each pixel depends on the samples not merely in that pixel, but in neighboring pixels as well. The pixels are sent to the display system to be put into a file or onto a framebuffer.

1.2.5 Motion Blur and Depth of Field

Interestingly, very few changes need to be made to the basic Reyes rendering pipeline to support several of the most interesting and unique features of *PRMan*. One of the most often used advanced features is motion blur. Any primitive may be motion blurred either by a moving transformation or by a moving deformation (or both). In the former case, the primitive is defined as a single set of control points with multiple transformation matrices; in the later case, the primitive actually contains multiple sets of control points. In either case, the moving primitive when diced becomes a moving grid, with positional data for the beginning and ending of the motion path, and eventually a set of moving micropolygons.

The only significant change to the main rendering pipeline necessary to support this type of motion is that bounding box computations must include the entire motion path of the object. The hidden-surface algorithm modifications necessary to handle motion blur are implemented using the *stochastic sampling* algorithm first described by Cook, et al. in 1984. The hidden surface algorithm's point sample locations are each augmented with a unique sample time. As each micropolygon is sampled, it is translated along its motion path to the position required for each sample's time.

PRMan only shades moving primitives at the start of their motion and only supports linear motion of primitives between their start and stop positions. This means that shaded micropolygons do not change color over time, and they leave constant-colored streaks across the image. This is incorrect, particularly with respect to lighting, as micropolygons will "drag" shadows or specular highlights around with them. In practice, this artifact is rarely noticed due to the fact that such objects are so blurry anyway.

Depth of field is handled in a very similar way. The specified lens parameters and the known focusing equations make it easy to determine how large the circle of confusion is for each primitive in the scene based on its depth. That value increases the bounding box for the primitive and for its micropolygons. Stochastically chosen lens positions are determined for each point sample, and the samples are appropriately jittered on the lens in order to determine which blurry micropolygons they see.

1.2.6 Shading Before Hiding

Notice that this geometric pipeline has a feature that few other renderers share: The shading calculations are done before the hidden-surface algorithm is run. In normal scanline renderers, polygons are depth-sorted, the visible polygons are identified, and those polygons are clipped to create "spans" that cover portions of a scanline. The end points of those spans are shaded and then painted into pixels. In ray-tracing renderers, pixel sample positions are turned into rays, and the objects that are hit by (and therefore visible from) these rays are the only things that are shaded. Radiosity renderers often resolve colors independently of a particular viewpoint but nonetheless compute object intervisibility as a prerequisite to energy transfer. Hardware Z-buffer algorithms do usually shade before hiding, as Reyes does, however they generally only compute true shading at polygon vertices, not at the interiors of polygons.

One of the significant advantages of shading before hiding is that displacement shading is possible. This is because the final locations of the vertices are not needed by the hider until after shading has completed, and therefore the shader is free to move the points around without the hider ever knowing. In other algorithms, if the shader moved the vertices after the hider had resolved surfaces, it would invalidate the hider's results.

The biggest disadvantage of shading before hiding is that objects are shaded before it is known whether they will eventually be hidden from view. If the scene has a large depth complexity, large amounts of geometry might be shaded and then subsequently covered over by objects closer to the camera. That would be a large waste of compute time. In fact, it is very common for this to occur in Z-buffer renderings of complicated scenes. This disadvantage is addressed in the enhanced algorithm described in Section 1.3.2.

1.2.7 Memory Considerations

In this pipeline, each stage of processing converts a primitive into a finer and more detailed version. Its representation in memory gets larger as it is split, diced, busted, and sampled. However, notice also that every primitive is processed independently and has no interaction with other primitives in the system. Even sibling subprimitives are handled completely independently. For this reason, the geometric database can be streamed through the pipeline just as a geometric database is streamed through typical Z-buffer hardware. There is no long-term storage or buffering of a global database (except for the queue of split primitives waiting to be bounded, which is rarely large), and therefore there is almost no memory used by the algorithm. With a single exception: the visible point lists.

As stated earlier, no visible point list can be processed until it is known that all of the primitives that cover its pixel have, in fact, been processed. Because the streaming version of Reyes cannot know that any given pixel is done until the last primitive is rendered, it must store all the visible point lists for the entire image until the very end. The visible point lists therefore contain a point-sampled representation of the *entire* geometric database, and consequently are quite large. Strike that. They are absolutely huge — many gigabytes for a typical high-resolution film frame. Monstrously humongous. As a result, the algorithm simply would not be usable if implemented in this way. Memory-sensitive enhancements are required to make the algorithm practical.

1.3 Enhanced Geometric Pipeline

The original Reyes paper recognized that the memory issue was a problem, even more so in 1985 than it is now. So it provided a mechanism for limiting memory use, and other mechanisms have been added since, which together make the algorithm much leaner than most other algorithms.

1.3.1 Bucketing

In order to alleviate the visible point memory problem, a modified Reyes algorithm recognizes that the key to limiting the overall size of the visible point memory is to know that certain pixels are done before having to process the entire database. Those pixels can then be finished and freed early. This is accomplished by dividing the image into small rectangular pixel regions, known as *buckets*, which will be processed one-by-one to completion before significant amounts of work occur on other buckets.

The most important difference in the pipeline is in the bounding step, which now also sorts the primitives based on which buckets they affect (that is, which buckets the bounding box overlaps). If a primitive is not visible in the current bucket-of-interest, it is put onto a list for the first bucket where it will matter and is thereby held in its most compact form until truly needed.

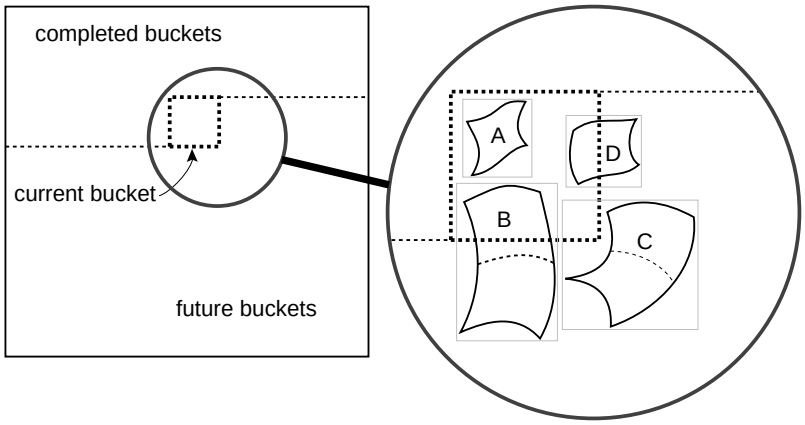


Figure 1.3: When the renderer processes the primitives that are on the list for the current bucket, their size and positions determine their fates.

After this, the algorithm proceeds in the obvious way. Buckets are processed one at a time. Objects are removed from the list for the current bucket and either split or diced. Split primitives might be added back to the list or might be added to the lists of future buckets, depending on their bounding boxes. Diced primitives go through the normal shading pipeline and are busted. During busting, the micropolygons are bound and similarly bucket-sorted. Micropolygons that are not in the current bucket-of-interest are not sampled until the appropriate bucket is being processed. Figure 1.3 shows four primitives whose disposition is different. Primitive A will be diced, shaded, and sampled in the current bucket. Primitive B needs to be split, and half will return to the current bucket while half will be handled in a future bucket. Primitive C is in the current bucket because its bounding box touches it (as shown), but once split, you can see that both child primitives will fall into future buckets. Primitive D will be diced and shaded in the current bucket, but some of the micropolygons generated will be held for sampling until the next bucket is processed.

Eventually, there are no more primitives in the current bucket's list, because they all have either been sampled or transferred to future buckets. At that point, all of the visible point lists in that bucket can be resolved and the pixels for that bucket displayed. This is why *PhotoRealistic RenderMan* creates output pixels in little blocks, rather than in scanlines like many algorithms. Each block is a bucket. The algorithm does not require that the buckets be processed in a particular order, but in practice the implementation still uses a scanline-style order, processing buckets one horizontal row at a time, left to right across the row, and rows from top to bottom down the image.

The major effect of this pipeline change is the utilization of memory. The entire database is now read into memory and sorted into buckets before any significant amount of rendering is done. The vast majority of the geometric database is stored in the relatively compact form of per-bucket lists full of high-level geometric primitives. Some memory is also used for per-bucket lists of micropolygons that have already been diced and shaded but are not relevant to the current bucket. The visible point lists have been reduced to only those that are part of the current bucket, a small fraction of the lists required for an entire image. Thus we have traded visible point list memory for geo-

metric database memory, and in all but the most pathological cases, this trade-off wins by orders of magnitude.

1.3.2 Occlusion Culling

As described so far, the Reyes algorithm processes primitives in arbitrary order within a bucket. In the preceeding discussion, we mentioned that this might put a primitive through the dicing/shading/hiding pipeline that will eventually turn out to be obscured by a later primitive that is in front of it. If the dicing and shading of these objects takes a lot of computation time (which it generally does in a photorealistic rendering with visually complex shaders), this time is wasted. As stated, this problem is not unique to Reyes (it happens to nearly every Z-buffer algorithm), but it is still annoying. The enhanced Reyes algorithm significantly reduces this inefficiency by a process known as *occlusion culling*.

The primitive bound-and-sort routine is changed to also sort each bucket's primitives by depth. This way, objects close to the camera are taken from the sorted list and processed first, while farther objects are processed later. Simultaneously, the hider keeps track of a simple hierarchical data structure that describes how much of the bucket has been covered by opaque objects and at what depths. Once the bucket is completely covered by opaque objects, any primitive which is entirely behind that covering is occluded. Because it cannot be visible, it can be culled before the expensive dicing and shading occurs (in the case of procedural primitives, before they are even loaded into the database). By processing primitives in front-to-back order, we maximize the probability that at least some objects will be occluded and culled. This optimization provides a two- to ten-times speedup in the rendering times of typical high-resolution film frames.

1.3.3 Network Rendering

In the enhanced Reyes algorithm, most of the computation — dicing, shading, hiding, and filtering — takes place once the primitives have been sorted into buckets. Moreover, except for a few details discussed later, those bucket calculations are generally independent of each other. For this reason, buckets can often be processed independently, and this implies that there is an opportunity to exploit parallelism. *PRMan* does this by implementing a large-grain multiprocessor parallelism scheme known as *NetRenderMan*.

With NetRenderMan, a parallelism-control client program dispatches work in the form of bucket requests to multiple independent rendering server processes. Server processes handle all of the calculation necessary to create the pixels for the requested bucket, then make themselves available for additional buckets. Serial sections of the code (particularly in database sorting), redundant work due to primitives that overlap multiple buckets, and network latency cut the overall multiprocessor efficiency to approximately 70-80% on typical frames, but nevertheless the algorithm often shows linear speedup through 8 to 10 processors. Because these processes run independently of each other, with no shared data structures, they can run on multiple machines on the network, and in fact on multiple processor architectures in a heterogeneous network, with no additional loss of efficiency.

1.4 Rendering Attributes and Options

With this background, it is easy to understand certain previously obscure rendering attributes and options, and why they affect memory and/or rendering time, and also why certain types of geometric models render faster or slower than others.

1.4.1 Shading Rate

In the RenderMan Interface, the `ShadingRate` of an object refers to the frequency with which the primitive must be shaded (actually measured by sample area in pixels) in order to adequately capture its color variations. For example, a typical `ShadingRate` of 1.0 specifies one shading sample per pixel, or roughly Phong-shading style. In the Reyes algorithm, this constraint translates into micropolygon size. During the dicing phase, an estimate of the raster space size of the primitive is made, and this number is divided by the shading rate to determine the number of micropolygons that must make up the grid. However, the dicing tessellation is always done in such a manner as to create (within a single grid) micropolygons that are of identically-sized rectangles in the parametric space of the primitive. For this reason, it is not possible for the resulting micropolygons in a grid to all be exactly the same size in raster space, and therefore they will only approximate the shading rate requested of the object. Some will be slightly larger, others slightly smaller than desired.

Notice, too, that any adjacent sibling primitive will be independently estimated, and therefore the number of micropolygons that are required for it may easily be different (even if the sibling primitive is the same size in parametric space). In fact, this is by design, as the Reyes algorithm fundamentally takes advantage of *adaptive subdivision* to create micropolygons that are approximately equal in size in raster space independent of their size in parametric space. That way, objects farther away from the camera will create a smaller number of equally sized micropolygons, instead of creating a sea of inefficient nanopolygons. Conversely, objects very close to the camera will create a large number of micropolygons, in order to cover the screen with sufficient shading samples to capture the visual detail that is required of the close-up view. For this reason, it is very common for two adjacent grids to have different numbers of micropolygons along their common edge (see Figure 1.4), and this difference in micropolygon size across an edge is the source of some shading artifacts that are described in Section 1.5.4.

In most other rendering algorithms, a shading calculation occurs at every hidden-surface sample, so raising the antialiasing rate increases the number of shading samples as well. Because the vast majority of calculations in a modern renderer are in the shading and hidden-surface calculations, increasing `PixelSamples` therefore has a direct linear effect on rendering time. In Reyes, these two calculations are decoupled, because shading rate affects only micropolygon dicing, not hidden surface evaluation. Antialiasing can be increased without spending any additional time shading, so raising the number of pixel samples in a Reyes image will make a much smaller impact on rendering time than in other algorithms (often in the range of percentage points instead of factors). Conversely, adjusting the shading rate will have a large impact on rendering time in images where shading dominates the calculation.

1.4.2 Bucket Size and Maximum Grid Size

The bucket size option obviously controls the number of pixels that make up a bucket and inversely controls the number of buckets that make up an image. The most obvious effect of this control is

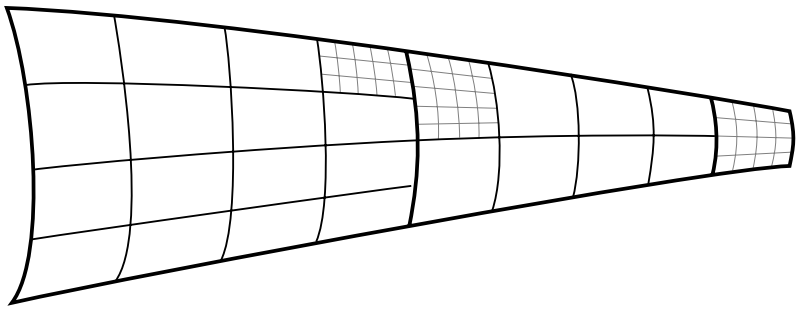


Figure 1.4: Adaptive parametric subdivision leads to adjacent grids that are different sizes parametrically and micropolygons that approximate the desired shading rate.

to regulate the amount of memory devoted to visible point lists. Smaller buckets are more memory efficient because less memory is devoted to visible point lists. Less obviously, smaller buckets partition the geometric database into larger numbers of shorter, sorted primitive lists, with some consequential decrease in sorting time. However, this small effect is usually offset by the increase in certain per-bucket overhead.

The maximum grid size option controls dicing by imposing an upper limit on the number of micropolygons that may occur in a single grid. Larger grids are more efficient to shade because they maximize vector pipelining. However, larger grids also increase the amount of memory that can be devoted to shader global and local variable registers (which are allocated in rectangular arrays the size of a grid). More interestingly, however, the maximum grid size creates a loose upper bound on the pixel area that a grid may cover on-screen — a grid is unlikely to be much larger than the product of the maximum grid size and the shading rate of the grid. This is important in relation to the bucket size because grids that are larger than a bucket will tend to create large numbers of micropolygons that fall outside of the current bucket and that must be stored in lists for future buckets. Micropolygons that linger in such lists can use a lot of memory.

In the past, when memory was at a premium, it was often extremely important to optimize the bucket size and maximum grid size to limit the potentially large visible point and micropolygon list memory consumption. On modern computers, it is rare that these data structures are sufficiently large to concern us, and large limits are perfectly acceptable. The default values for bucket size, 16×16 pixel buckets, and maximum grid size, 256 micropolygons per grid, work well except under the most extreme situations.

1.4.3 Transparency

Partially transparent objects cause no difficulty to the algorithm generally; however, they can have two effects on the efficiency of the implementation. First, transparent objects clearly affect the memory consumption of the visible point lists. Due to the mathematical constraints of the compositing algebra used by *PRMan*, it is not possible to composite together the various partially transparent layers that are held in the visible-point list of a sample until the sample is entirely complete. Notice that an opaque layer can immediately truncate a list, but in the presence of large amounts of

transparency, many potentially visible layers must be kept around. Second, and more importantly, transparent layers do not contribute to the occlusion culling of future primitives, which means that more primitives are diced and shaded than usual. Although this should be obvious (since those primitives are probably going to be seen through the transparent foreground), it is often quite surprising to see the renderer slow down as much as it does when the usually extremely efficient occlusion culling is essentially disabled by transparent foreground layers.

1.4.4 Displacement Bounds

Displacement shaders can move grid vertices, and there is no built-in constraint on the distance that they can be moved. However, recall that shading happens halfway through the rendering pipeline, with bounding, splitting, and dicing happening prior to the evaluation of those displacements. In fact, the renderer relies heavily on its ability to accurately yet tightly bound primitives so that they can be placed into the correct bucket. If a displacement pushes a grid vertex outside of its original bounding box, it will likely mean that the grid is also in the wrong bucket. Typically, this results in a large hole in the object corresponding to the bucket where the grid "should have been considered, but wasn't."

This is avoided by supplying the renderer a bound on the size of the displacement generated by the shader. From the shader writer's point of view, this number represents the worst-case displacement magnitude — the largest distance than any vertex might travel, given the calculations inherent in the displacement shader itself. From the renderer's point of view, this number represents the padding that must be given to every bounding-box calculation prior to shading, to protect against vertices leaving their boxes. The renderer grows the primitive bounding box by this value, which means that the primitive is diced and shaded in a bucket earlier than it would normally be processed. This often leads to micropolygons that are created long before their buckets need them, which then hang around in bucket micropolygon lists wasting memory, or primitives that are shaded before it is discovered that they are off-screen. Because of these computational and memory inefficiencies of the expanded bounds, it is important that the displacement bounds be as tight as possible, to limit the damage.

1.4.5 Extreme Displacement

Sometimes the renderer is stuck with large displacement bounds, either because the object really does displace a large distance or because the camera is looking extremely closely at the object and the displacements appear very large on-screen. In extreme cases, the renderer can lose huge amounts of memory to micropolygon lists that contain most of the geometric database. In cases such as these, a better option is available. Notice that the problem with the displacement bound is that it is a worst-case estimate over the primitive as a whole, whereas the small portion of the primitive represented by a single small grid usually does not contain the worst-case displacement and actually could get away with a much smaller (tighter) bound. The solution to this dilemma is to actually *run the shader* to evaluate the true displacement magnitude for each grid on a grid-by-grid basis, and then store those values with the grid as the *exact* displacement bound. The disadvantage of this technique is that it requires the primitive to be shaded twice, once solely to determine the displacement magnitude and then again later to generate the color when the grid is processed normally in its new bucket. Thus, it is a simple space-time trade-off.

This technique is enabled by the `extremedisplacement` attribute, which specifies a threshold raster distance. If the projected raster size of the displacement bound for a primitive exceeds the extreme displacement limit for that primitive, the extra shading calculations are done to ensure economy of memory. If it does not, then the extra time is not spent, under the assumption that for such a small distance the memory usage is transient enough to be inconsequential.

1.4.6 Motion-Factor

When objects move quickly across the screen, they become blurry. Such objects are indistinct both because their features are spread out over a large region, and because their speed makes it difficult for our eyes to track them. As a result, it is not necessary to shade them with particularly high fidelity, as the detail will just be lost in the motion blur. Moreover, every micropolygon of the primitive will have a very large bounding box (corresponding to the length of the streak), which means that fine tessellations will lead to large numbers of micropolygons that linger a long time in memory as they are sampled by the many buckets along their path.

The solution to this problem is to enlarge the shading rate of primitives if they move rapidly. It is possible for the modeler to do this, of course, but it is often easier for the renderer to determine the speed of the model and then scale the shading rates of each primitive consistently. The attribute that controls this calculation is a `GeometricApproximation` flag known as `motionfactor`. For obscure reasons, `motionfactor` gives a magnification factor on shading rate per every 16 pixels of blurring. Experience has shown that a motion-factor of 1.0 is appropriate for a large range of images.

The same argument applies equally to depth-of-field blur, and in the current implementation, `motionfactor` (despite its name) also operates on primitives with large depth-of-field blurs as well.

1.5 Rendering Artifacts

Just as an in-depth understanding of the Reyes pipeline helps you understand the reason for, and utility of, various rendering options and attributes, it also helps one understand the causes and solutions for various types of geometric rendering artifacts that can occur while using *PhotoRealistic RenderMan*.

1.5.1 Eye Splits

Sometimes *PhotoRealistic RenderMan* will print the error message “Cannot split primitive at eye plane,” usually after appearing to stall for quite a while. This error message is a result of perhaps the worst single algorithmic limitation of the modified Reyes algorithm: In order to correctly estimate the shading rate required for a primitive, the primitive must first be projected into raster space in order to evaluate its size. Additionally, recall that the first step in the geometric pipeline is to bound the primitive and sort it into buckets based on its position in raster space, which requires the same projection. The problem is that the mathematics of perspective projection only works for positions that are in front of the camera. It is not possible to project points that are behind the camera. For this reason, the renderer must divide the primitive into areas that are in front of and areas that are behind the camera.

Most rendering algorithms, if they require this projection at all, would clip the primitive against the near clipping plane (hence the name) and throw away the bad regions. However, the entire Reyes geometric and shading pipelines require subprimitives that are rectangular in parametric space, which can be split and diced cleanly. Clipping does not create such primitives and cannot be used. Instead, Reyes simply splits the primitive hoping that portions of the smaller subprimitives will be easier to classify and resolve.

Figure 1.5 shows the situation. Notice that primitives which lie entirely forward of the eye plane are projectable, so can be accepted. Primitives which lie entirely behind the near clipping plane can be trivially culled. It is only primitives which span both planes that cannot be classified and are split. The region between the planes can be called the "safety zone." If a split line lies entirely within this zone (in 3D, of course), both children of the bad primitive are classifiable, which is the situation we are hoping for. If the split line straddles a plane, at least we will shave some of the bad primitive away and the remaining job, we hope, is slightly easier. Reyes splits as smartly as it can, attempting to classify subprimitives.

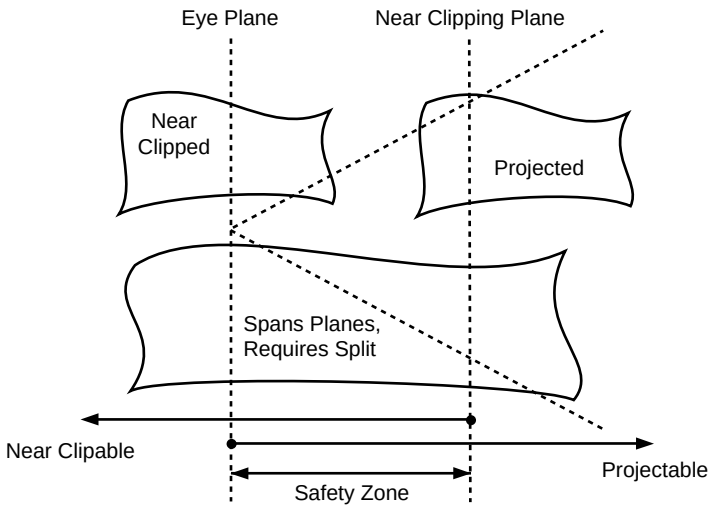


Figure 1.5: The geometric relationship of the near and eye planes gives rise to three categories of primitives: the cullable, the projectable, and the "spanners," which require splitting.

Unhappily, there are geometric situations where the splitting simply doesn't work in a reasonable number of steps. "Reasonable" is defined as a small integer because each split doubles the number of subprimitives, so even 10 attempts creates $2^{10} = 1024$ primitives. If, after splitting the maximum permitted number of times, the primitive still cannot be classified, Reyes gives up and throws the primitive away and prints the "Cannot split" message. If that primitive was supposed to be visible, the lost section will leave a hole in the image.

Primitives that have large displacement bounds, or are moving rapidly toward the camera, will exacerbate the eye-splitting problem because the parametric splitting process will do very little to reduce the bounding box. Indeed, for primitives for which the camera is inside the displacement

bound of part of the surface, or primitives whose motion path actually goes through the camera, splitting can never succeed.

In order to reduce the artifacts due to eye-split culling, the key is to give the renderer the largest possible safety zone. Place the near clipping plane as far forward as is possible without otherwise affecting the image. The near clipping plane can be placed surprisingly far forward for most shots made with cameras that have reasonable fields of view. If you don't normally set the clipping plane, set it to some small but reasonable value immediately — the default value of `1e-10` is just about as bad as you could get! The number of splitting iterations that will be permitted can be controlled with a rendering option, and if you permit a few more, you can sometimes help handling of large but otherwise simple primitives. Beware, however, the displacement and motion cases, because upping the limit will just let the renderer waste exponentially more time before it gives up.

Also, make sure that displacement bounds for primitives near the camera (for example, the ground plane) are as tight as possible and that the shader is coded so that the displacement itself is as small as possible. If you place the camera so close to some primitive that the displacement bound is a significant portion of the image size, there will be no end of trouble with both eye-splits and displacement stretching (discussed later). In fly-overs of the Grand Canyon, the Canyon should be modeled, not implemented as a displacement map of a flat plane!

It will also help to keep the camera as high off the ground as possible without ruining the composition of the shot. Reyes simply has lots of trouble with worm's-eye views. And make sure that no object is flying through the camera (you wouldn't do that in live-action photography, would you?). Generally, if you pretend that the CG camera has a physical lens that keeps objects in the scene at least a certain distance away and respect that border, you will have fewer problems with eye-splits.

1.5.2 Patch Cracks

Patch cracks are tiny holes in the surface of objects that are caused by various errors in the approximation of primitives by their tessellations (we'll use the term loosely to include tessellation-created cracks on primitives other than patches). Patch cracks usually appear as scattered pinholes in the surface, although they can sometimes appear as lines of pinholes or as small slits. Importantly, they always appear along parametric lines of primitives. They are recognizably different from other holes created by clipping, culling, or bucketing errors, which are usually larger, often triangular, and occur in view-dependent places (like on silhouettes or aligned with bucket boundaries).

Patch cracks occur because when objects are defined by sets of individual primitives, the connectedness of those primitives is only implied by the fact that they abut, that they have edges which have vertices in common. There is no way in the RenderMan Interface to explicitly state that separate primitives have edges that should be "glued together." Therefore, as the primitives go through the geometric pipeline independently, there is a chance that the mathematical operations that occur on one version of the edge will deviate from those on the other version of the edge, and the vertices will diverge. If they do, a crack occurs between them.

The deviations can happen in several ways. One is the tessellation of the edge by adjacent grids being done with micropolygons of different sizes. Figure 1.6 shows that such tessellations naturally create intermediate grid vertices that do not match, and there are tiny holes between the grids. For many years, *PRMan* has had a switch that eliminated most occurrences of this type of crack. Known as *binary dicing*, it requires that every grid have tessellations which create a power-of-two number of micropolygons along each edge. Although these micropolygons are smaller than are required by shading rate alone, binary dicing ensures that adjacent grids have tessellations that are powers of two

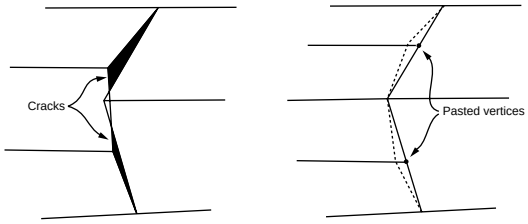


Figure 1.6: Tessellation differences result in patch cracks.

multiples of each other (generally, a single factor of two). Thus, alternating vertices will coincide, and the extra vertices on one side are easily found and "pasted" to the other surface.

Another way that patch cracks can happen is when the displacement shader that operates on grid vertices gets different results on one grid from another. If a common vertex displaces differently on two grids, the displacement literally rips the surface apart, leaving a crack between. One common reason for such differing results is displacement mapping using texture filter sizes that are mismatched (see Section 1.5.4). The texture call then returns a slightly different value on the two grids, and one grid displaces to a different height than the other. Another common reason is displacement occurring along slightly different vectors. For example, the results of `calculatenormal` are almost guaranteed to be different on the left edge of one grid and on the right edge of the adjacent grid. If displacement occurs along these differing vectors, the vertices will obviously go to different places, opening a crack. Unfortunately, only careful coding of displacement shaders can eliminate this type of cracking.

Notice that patch cracks cannot happen on the interiors of grids. Because grid micropolygons explicitly share vertices, it is not possible for such neighbor micropolygons to have cracks between them. For this reason, patch cracks will only occur along boundaries of grids, and therefore along parametric edges. In *PRMan* versions prior to 3.9, patch cracks could occur on any grid edge, including those that resulted from splitting a patch into subpatches. Later versions of *PRMan* have a crack-avoidance algorithm that glues all such subpatches together. Therefore, modern versions of *PRMan* will only exhibit patch cracks along boundaries of original primitives, not along arbitrary grid boundary.

1.5.3 Displacement Stretching

Another problem that displacement shaders might create is stretching of micropolygons. This is caused when displacement shaders move the vertices of a micropolygon apart, so that it no longer obeys the constraint that it is approximately the size specified by the shading rate.

In the process of dicing, the renderer estimates the size of the primitive on-screen and makes a grid that has micropolygons that approximately match the shading rate. Shading then occurs on the vertices of the grid (the corners of the micropolygons). Displacement shaders are permitted to move grid vertices, but there is no constraint on where they are moved. If two adjacent grid vertices are moved in different directions (wildly or subtly), the area of the micropolygon connecting them will change. Generally, this change is so small that the micropolygon is still safely in the range expected of shading rate. However, if the displacement function has strong high frequencies, adjacent grid

vertices might move quite differently. For example, an embossing shader might leave some vertices alone while moving vertices inside the embossed figure quite a distance. The micropolygons whose corners move very differently will change size radically, and sometimes will be badly bent or twisted (see Figure 1.7).

Twisted micropolygons have unusual normal vectors, and this alone may be enough to cause shading artifacts. For example, highly specular surfaces are very sensitive to normal vector orientation, so a micropolygon that is twisted in an unusual direction may catch an unexpected highlight.

More common, however, is that the stretching of the micropolygons will itself be visible in the final image. An individual flat-shaded micropolygon creates a constant-colored region in the image. With a standard shading rate of around a pixel, every pixel gets a different micropolygon and the flat shading is not visible. But large stretched or long twisted micropolygons will cover many pixels, and the constant-colored region will be evident. Sometimes this takes the form of alternating dark and light triangles along the face of a displacement "cliff." Corners of micropolygons that are shaded for the top of the plateau hang down, while corners of micropolygons that are shaded for the valley poke up, interleaved like teeth of a gear.

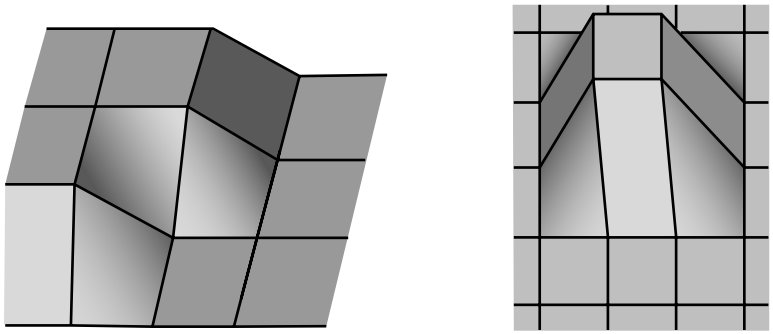


Figure 1.7: Displacement stretching leads to micropolygons that are bent, twisted, or significantly larger than anticipated.

The visual artifacts of these problems can be somewhat ameliorated by using smooth shading interpolation, which will blur the shading discontinuities caused by the varying normals. However, the geometric problems remain. The primary solution is to lower the frequency content of the displacement shader so that adjacent micropolygons cannot have such wildly varying motion. (See Chapter 11 of *Advanced RenderMan: Creating CGI for Motion Pictures* (Shader Antialiasing) for hints.) If this cannot be done, the brute-force approach is to reduce the shading rate to values such as 0.25 pixels or smaller so that even stretched micropolygons stay under 1 pixel in size. However, this will have significant performance impact, because shading time is inversely proportional to shading

rate. Fortunately, shading rate is an attribute of individual primitives, so the extra expense can be limited to the part of the model that requires it.

1.5.4 Texture Filter Mismatches

As primitives are split, their subprimitives proceed through the rendering pipeline independently, and when the time comes to dice them, their own individual size on-screen determines the tessellation rate. As a result, it is often the case that the adjacent grids resulting from adjacent subprimitives will project to different sizes on the screen, and as a result will tessellate at different rates during dicing. Tessellated micropolygons are rectangular in the parametric space of the original primitive, and all of the micropolygons in a single grid will be the same size in that parametric space, but due to to the differing tessellation rates, the micropolygons that make up the adjacent grids will have different sizes in parametric space.

One of the artifacts that results from this difference is that filtering calculations based on parametric size will change discontinuously across a grid boundary. In Chapter 11 of *Advanced RenderMan*, various filtering techniques have been discussed that use parametric size as part of the calculation of filter width. This type of size discontinuity will result in filtering discontinuities, which can be visible in the final image. For example, in simple texturing, the texture filter size defaults to the micropolygon size. The resulting texture can have visible changes in sharpness over the surface of an otherwise smooth primitive. If the result of a texture call is used as a displacement magnitude, a displacement crack can result (Section 1.5.2).

Recent versions of *PRMan* have significantly reduced problems such as these by the introduction of smooth derivatives. The derivatives and the parametric size values that are available to shaders now describe a smoothly varying idealized parametric size for the micropolygon. That is, the values do not exactly match the true size of the micropolygon in parametric space, but instead track closely the desired parametric size given the shading rate requested (in some sense compensating for the compromises that needed to be made to accommodate binary dicing or other tessellation constraints at the time of dicing). These smoothly varying parametric size estimates ameliorate texture filter size mismatches, both in built-in shading functions and in antialiased procedural textures. Generally, the fact that micropolygons are not exactly the size that they advertise is a small issue, and where it is an issue, it can be compensated for by minor modifications to the shader.

1.5.5 Conclusion

The Reyes rendering architecture is so general and flexible that new primitives, new graphics algorithms, and new effects features have been added modularly to the existing structure almost continously for over 15 years. The system has evolved from an experimental testbed with a seemingly unattainable dream of handling tens of thousands of primitives into a robust production system that regularly handles images a hundred times more complex than that. The speed and memory enhancements that have been added may appear to have been short-term requirements, as Moore's law allows us to run the program on computers that are faster and have more memory without any additional programming. However, this is shortsighted, for our appetite for complexity has scaled, too, as fast or faster than Moore's Law allows. Undoubtedly, in 15 more years, when computers with a terabyte of main memory are common and optical processors chew up 1 billion primitives without flinching, we will still be using the Reyes architecture to compute our holofilms.

Chapter 2
The Secret Life of Lights and Surfaces

Larry Gritz
Pixar Animation Studios
lg@pixar.com

Abstract

In this section we will explore how surfaces respond to light, as well as how light sources themselves operate, which are critical aspects of the overall appearance of materials. In doing so, we will build up a variety of library routines that implement different material appearances and light properties.[1]

2.1 Built-in local illumination models

You've probably seen countless shaders that ended with the following lines of code, or something remarkably close:

```
Ci = Ct * (Ka*ambient() + Kd*diffuse(Nf)) +
     specularcolor * Ks*specular(Nf,-normalize(I),roughness);
Oi = Os;  Ci *= Oi;
```

Three functions are used here that have not been previously described:

`color diffuse(vector N)`

> Calculates light widely and uniformly scattered as it bounces from a light source off of the surface. Diffuse reflectivity is generally approximated by Lambert's law:
>
> $$\sum_{i=1}^{\mathrm{nlights}} Cl_i \max(0, N \cdot L_i)$$

[1] This section is substantially reprinted from *Advanced RenderMan: Creating CGI for Motion Pictures*, by Anthony A. Apodaca and Larry Gritz, ISBN 1-55860-618-1, published by and copyright ©2000 by Morgan Kaufmann Publishers, all rights reserved, and is used with permission. Hey! Why don't you go get the whole book instead of reading a couple out-of-context excerpts?

where for each of the i light sources, L_i is the unit vector pointing toward the light, Cl_i is the light color, and N is the unit normal of the surface. The $\max$ function ensures that lights with $N \cdot L_i < 0$ (i.e., those *behind* the surface) do not contribute to the calculation.

`color specular(vector N, V; float roughness)`

Computes so-called *specular* lighting, which refers to the way that glossier surfaces have noticeable bright spots or highlights resulting from the narrower (in angle) scattering of light off the surface. A typical formula for such scattering might be the *Blinn-Phong* model:

$$\sum_{i=1}^{\mathrm{nlights}} Cl_i \max(0, N \cdot H)^{1/\mathrm{roughness}}$$

where H is the vector halfway between the viewing direction and the direction of the light source (i.e., `normalize(normalize(-I)+normalize(L))`).

The equation above is for the Blinn-Phong reflection model, which is what is dictated by the *RenderMan Interface Specification*. *PRMan* actually uses a slightly different, proprietary formula for `specular()`. *BMRT* also uses a slightly nonstandard formulation of `specular()` in order to more closely match *PRMan*. So beware — though the spec dictates Blinn-Phong, individual implementations can and do substitute other reflection models for `specular()`.

`color ambient()`

Returns the contribution of so-called *ambient* light, which comes from no specific location but rather represents the low level of scattered light in a scene after bouncing from object to object.[2]

Therefore those three lines we had at the end of our shaders calculate a weighted sum of ambient, diffuse, and specular lighting components. Typically, the diffuse and ambient light is filtered by the base color of the object, but the specular contribution is not (or is filtered by a separate `specularcolor`).

We usually assign `Oi`, the opacity of the surface, to simply be the default surface opacity `Os`. Finally, we scale the output color by the output opacity, because RenderMan requires shaders to compute premultiplied opacity values.

When `specularcolor` is 1 (i.e., white), these calculations yield an appearance closely resembling plastic. Let us then formalize it with the Shading Language function `MaterialPlastic` (in Listing 2.1).

With this function in our library, we could replace the usual ending lines of our shader with:

```
Ci = MaterialPlastic (Nf, V, Cs, Ka, Kd, Ks, roughness);
Oi = Os;  Ci *= Oi;
```

[2] In most renderers, ambient light is typically approximated by a low-level, constant, non-directional light contribution set by the user in a rather ad hoc manner. When renderers try to accurately calculate this interreflected light in a principled manner, it is known as *global illumination*, or, depending on the exact method used, as *radiosity*, *path tracing*, *Monte Carlo integration*, and others.

Listing 2.1 `MaterialPlastic` computes a local illumination model approximating the appearance of ordinary plastic.

```
/* Compute the color of the surface using a simple plastic-like BRDF.
 * Typical values are Ka=1, Kd=0.8, Ks=0.5, roughness=0.1.
 */
color MaterialPlastic (normal Nf;  color basecolor;
                       float Ka, Kd, Ks, roughness;)
{
    extern vector I;
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf))
           + Ks*specular(Nf,-normalize(I),roughness);
}
```

For the remainder of this chapter, functions that compute completed material appearances will be named with the prefix `Material`, followed by a description of the material family. Arguments passed will generally include a base color, surface normal, viewing direction, and a variety of weights and other knobs that select individual appearances from the family of materials.

The implementation of the `Material` functions will typically be to compute a weighted sum of several *primitive local illumination functions*. Before long, it will be necessary to move beyond `ambient()`, `diffuse()`, and `specular()` to other, more sophisticated local illumination functions. When we start writing our own local illumination functions, we will use the convention of naming them with the prefix `LocIllum` and will typically name them after their inventors (e.g., `LocIllumCookTorrance`).

But first, let us see what effects we can get with just the built-in `specular()` and `diffuse()` functions.

2.1.1 Matte Surfaces

The typical combination of the built-in `diffuse()` and `specular()` functions, formalized in `MaterialPlastic()`, is great at making materials that look like manufactured plastic (such as toys). But many objects that you model will not be made from materials that feature a prominent specular highlight. One could, of course, simply call `MaterialPlastic()` passing `Ks` = 0. However, that seems wasteful to call the potentially expensive `specular()` function only to multiply it by zero. Our solution is to create a separate, simpler `MaterialMatte` that only calculates the ambient and Lambertian (via `diffuse()`) contributions without a specular highlight, as shown in Listing 2.2.

Listing 2.2 `MaterialMatte` computes the color of the surface using a simple Lambertian BRDF.

```
color MaterialMatte (normal Nf;  color basecolor;  float Ka, Kd;)
{
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf));
}
```

2.1.2 Rough metallic surfaces

Another class of surfaces not well modeled by the `MaterialPlastic` function is that of metals. Deferring until the next section metals that are polished to the point that they have visible reflections of surrounding objects, we will concentrate for now on roughened metallic surfaces without coherent reflections.

Cook and Torrance (Cook and Torrance, 1981; Cook and Torrance, 1982) realized that an important difference between plastics and metals is the effect of the base color of the material on the specular component. Many materials, including paint and colored plastic, are composed of a transparent substrate with embedded pigment particles (see Figure 2.1). The outer, clear surface boundary both reflects light specularly (without affecting its color) and transmits light into the media that is permeated by pigment deposits. Some of the transmitted light is scattered back diffusely after being filtered by the pigment color. This white highlight contributes greatly to the perception of the material as plastic.

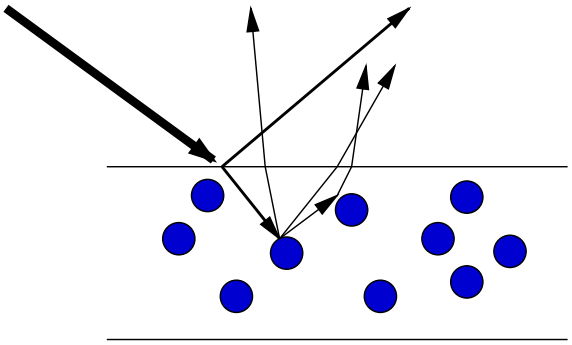


Figure 2.1: Cross section of a plastic-like surface.

Homogeneous materials, including metals, lack a transparent outer layer that would specularly reflect light without attenuating its color. Therefore, in these materials, all reflected light (including specular) is scaled by the material's base color. This largely contributes to the metallic appearance. We implement this look in `MaterialRoughMetal` (Listing 2.3).

2.1.3 Backlighting

So far, the materials we have simulated have all been assumed to be of substantial thickness. In other words, lights on the same side of the surface as the viewer reflect off the surface and are seen by the camera, but lights "behind" the surface (from the point of view of the camera) do not scatter light around the corner so that it contributes to the camera's view.

It is not by accident that such backlighting is excluded from contributing. Note that the `diffuse()` function takes the surface normal as an argument. The details of its working will be revealed in Section 2.3; let us simply note here that this normal parameter is used to exclude the contribution of light sources that do not lie on the same side of the surface as the viewer.

But thinner materials — paper, lampshades, blades of grass, thin sheets of plastic — do have appearances that are affected by lights behind the object. These objects are translucent, so lights lights

Listing 2.3 `MaterialRoughMetal` calculates the color of the surface using a simple metal-like BRDF.

```
/* Compute the color of the surface using a simple metal-like BRDF.  To
 * give a metallic appearance, both diffuse and specular components are
 * scaled by the color of the metal.  It is recommended that Kd < 0.1,
 * Ks > 0.5, and roughness > 0.15 to give a believable metal-
lic appearance.
 */
color MaterialRoughMetal (normal Nf;  color basecolor;
                          float Ka, Kd, Ks, roughness;)
{
    extern vector I;
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf) +
                        Ks*specular(Nf,-normalize(I),roughness));
}
```

Figure 2.2: The finishes (left to right) `MaterialPlastic`, `MaterialRoughMetal`, and `MaterialMatte` applied to a vase.

shine *through* the object, albeit usually at a lower intensity than the reflections of the lights in front of the object. Note the difference between *translucency*, the diffuse transmission of very scattered light through a thickness of material, and *transparency*, which means one can see a coherent image through the object. Ordinary paper is translucent, whereas glass is transparent. Therefore, since

```
diffuse(Nf)
```

sums the Lambertian scattering of lights on the viewer's side of the surface, then the lights from the *back side* should be described by

```
diffuse(-Nf)
```

In fact, this works exactly as we might hope. Thus, making a material translucent is as easy as adding an additional contribution of `diffuse()` oriented in the backwards direction (and presumably with a different, and smaller, weight denoted by `Kt`). This is exemplified by the `MaterialThinPlastic` function of Listing 2.4.

Listing 2.4 `MaterialThinPlastic` implements a simple, thin, plastic-like BRDF.

```
/* Compute the color of the surface using a simple, thin, plastic-
like BRDF.
 * We call it _thin_ because it includes a transmission compo-
nent to allow
 * light from the _back_ of the surface to affect the appear-
ance.  Typical
 * values are Ka=1, Kd=0.8, Kt = 0.2, Ks=0.5, roughness=0.1.
 */
color MaterialThinPlastic (normal Nf;  vector V;  color basecolor;
                           float Ka, Kd, Kt, Ks, roughness;)
{
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf) + Kt*diffuse(-Nf))
           + Ks*specular(Nf,V,roughness);
}
```

2.2 Reflections

We have covered materials that are linear combinations of Lambertian diffuse and specular components. However, many surfaces are polished to a sufficient degree that one can see coherent reflections of the surrounding environment. This section will discuss two ways of simulating this phenomena and show several applications.

People often assume that mirror-like reflections require ray tracing. But not all renderers support ray tracing (and, in fact, those renderers are typically much faster than ray tracers). In addition, there are situations where even ray tracing does not help. For example, if you are compositing a CG object into a live-action shot, you may want the object to reflect its environment. This is not possible even with ray tracing, because the environment does not exist in the CG world. Of course, one could laboriously model all the objects in the live-action scene, but this seems like too much work for a few reflections.

Luckily, RenderMan Shading Language provides support for faking these effects with texture maps, even for renderers that do not support any ray tracing. In this case, we can take a multipass approach, first rendering the scene from the points of view of the reflectors, then using these first passes as special texture maps when rendering the final view from the main camera.

2.2.1 Environment Maps

Environment maps take images of six axis-aligned directions from a particular point (like the six faces of a cube) and allow you to look up texture on those maps, indexed by a direction vector, thus simulating reflection. An example of an "unwrapped" environment map is shown in Figure 2.3.

Accessing an environment map from inside your shader is straightforward with the built-in `environment` function:

```
type environment (string filename, vector R, ...)
```

The `environment()` function is quite analogous to the `texture()` call in several ways:

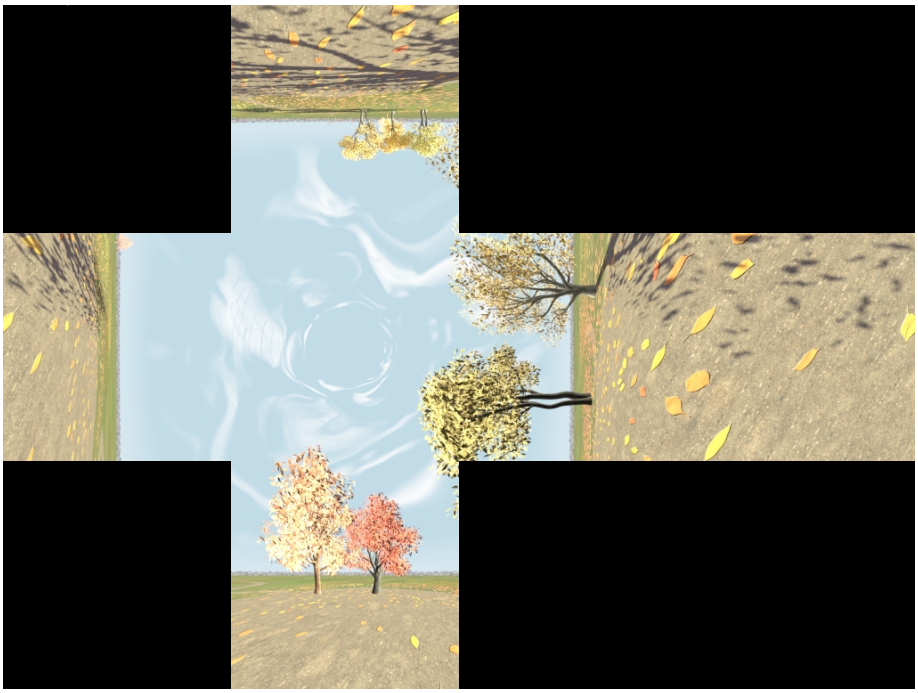


Figure 2.3: An example environment map.

- The return type can be explicitly cast to either `float` or `color`. If you do not explicitly cast the results, the compiler will try to infer the return type, which could lead to ambiguous situations.

- A `float` in brackets immediately following the filename indicates a starting channel (default is to start with channel 0).

- For environment maps, the texture coordinates consist of a direction vector. As with `texture()`, derivatives of this vector will be used for automatic filtering of the environment map lookup. Optionally, four vectors may be given to bound the angle range, and in that case no derivatives will be taken.

- The `environment()` function can take the optional arguments `"blur"`, `"width"`, and `"filter"`, which perform the same functions as for `texture()`.

Environment maps typically sample the mirror direction, as computed by the Shading Language built-in function `reflect()`. For example,

```
normal Nf = normalize (faceforward (N, I));
vector R = normalize (reflect (I, N));
color Crefl = color environment (envmapname, R);
```

Note that the `environment()` is indexed by direction only, not position. Thus, not only is the environment map created from the point of view of a single location, but all lookups are also made from that point. Alternatively, one can think of the environment map as being a reflection of a cube of infinite size. Either way, two points with identical mirror directions will look up the same direction in the environment map. This is most noticeable for flat surfaces, which tend to have all of their points index the same spot on the environment map. This is an obvious and objectionable artifact, especially for surfaces like floors, whose reflections are *very* sensitive to position.

We can partially overcome this difficulty with the following strategy:

1. We assume that the environment map exists on the interior of a sphere with a *finite* and known radius as well as a known center. Example: if the environment map is of a room interior, we choose a sphere radius representative of the room size. (Even if we have assembled an environment map from six rectangular faces, because it is indexed by direction only, for simplicity we can just as easily think of it as a spherical map.)

2. Instead of indexing the environment by direction only, we define a ray using the position and mirror direction of the point, then calculate the intersection of this ray with our aforementioned environment sphere.

3. The intersection with the environment sphere is then used as the environment lookup direction.

Thus, if a simple `environment()` lookup is like ray tracing against a sphere of infinite radius, then the scheme above is simply ray tracing against a sphere of a radius appropriate to the actual scene in the environment map.

As a subproblem, we must be able to intersect an environment sphere with a ray. A general treatment of ray/object intersections can be found in (Glassner, 1989), but the ray/sphere case is particularly simple. If a ray is described by end point E and unit direction vector I (expressed in the coordinate system of a sphere centered at its local origin and with radius r), then any point along the ray can be described as $E + It$ (for free parameter t). This ray intersects the sphere anyplace that $|E + It| = r$. Because the length of a vector is the square root of its dot product with itself, then

$$(E + It) \cdot (E + It) = r^2$$

Expanding the x, y, and z components for the dot product calculation yields

$$\begin{aligned} (E_x + I_x t)^2 + (E_y + I_y t)^2 + (E_z + I_z t)^2 - r^2 &= 0 \\ E_x^2 + 2E_x I_x t + I_x^2 t^2 + E_y^2 + 2E_y I_y t + I_y^2 t^2 & \\ + E_z^2 + 2E_z I_z t + I_z^2 t^2 - r^2 &= 0 \\ (I \cdot I)t^2 + 2(E \cdot I)t + E \cdot E - r^2 &= 0 \end{aligned}$$

for which the value(s) of t can be solved using the quadratic equation. This solution is performed by the `raysphere` function, which in turn is used by our enhanced `Environment` routine (see Listing 2.5). Note that `Environment` also returns an `alpha` value from the environment map, allowing us to composite multiple environment maps together.

Listing 2.5 `Environment` function replaces a simple `environment` call, giving more accurate reflections by tracing against an environment sphere of finite radius.

```
/* raysphere - calculate the intersection of ray (E,I) with a sphere
 * centered at the origin and with radius r.  We return the number of
 * intersections found (0, 1, or 2), and place the distances to the
 * intersections in t0, t1 (always with t0 <= t1).  Ignore any hits
 * closer than eps.
 */
float
raysphere (point E; vector I;   /* Origin and unit direction of the ray */
           float r;             /* radius of sphere */
           float eps;           /* epsilon - ignore closer hits */
           output float t0, t1; /* distances to intersection */
    )
{
    /* Set up a quadratic equation -- note that a==1 if I is normalized */
    float b = 2 * ((vector E) . I);
    float c = ((vector E) . (vector E)) - r*r;
    float discrim = b*b - 4*c;
    float solutions;
    if (discrim > 0) {           /* Two solutions */
        discrim = sqrt(discrim);
        t0 = (-discrim - b) / 2;
        if (t0 > eps) {
            t1 = (discrim - b) / 2;
            solutions = 2;
        } else {
            t0 = (discrim - b) / 2;
            solutions = (t0 > eps) ? 1 : 0;
        }
    } else if (discrim == 0) {  /* One solution on the edge! */
        t0 = -b/2;
        solutions = (t0 > eps) ? 1 : 0;
    } else {                    /* Imaginary solution -> no intersection */
        solutions = 0;
    }
    return solutions;
}


/* Environment() - A replacement for ordinary environment() lookups, this
 * function ray traces against an environment sphere of known, finite
 * radius.  Inputs are:
 *    envname - filename of environment map
 *    envspace - name of space environment map was made in
 *    envrad - approximate supposed radius of environment sphere
 *    P, R - position and direction of traced ray
 *    blur - amount of additional blur to add to environment map
 * Outputs are:
 *    return value - the color of incoming environment light
 *    alpha - opacity of environment map lookup in the direction R.
 * Warning -  the environment call itself takes derivatives, causing
 * trouble if called inside a loop or varying conditional!  Be cautious.
 */
color Environment ( string envname, envspace;  uniform float envrad;
                    point P;  vector R;  float blur; output float alpha;)
{
    /* Transform to the space of the environment map */
    point Psp = transform (envspace, P);
    vector Rsp = normalize (vtransform (envspace, R));
    uniform float r2 = envrad * envrad;
    /* Clamp the position to be *inside* the environment sphere */
    if ((vector Psp).(vector Psp) > r2)
        Psp = point (envrad * normalize (vector Psp));
    float t0, t1;
    if (raysphere (Psp, Rsp, envrad, 1.0e-4, t0, t1) > 0)
        Rsp = vector (Psp + t0 * Rsp);
    alpha = float environment (envname[3], Rsp, "blur", blur, "fill", 1);
    return color environment (envname, Rsp, "blur", blur);
}
```

2.2.2 Creating Cube Face Environment Maps

The creation of cube face environment maps is straightforward. First, six reflection images are created by rendering the scene from the point of view of the reflective object in each of six orthogonal directions in `"world"` space, given in Table 2.1.

Table 2.1: The six view directions in an environment map.

Face view	Axis toward top	Axis toward right
px (positive x)	$+y$	$-z$
nx (negative x)	$+y$	$+z$
py	$-z$	$+x$
ny	$+z$	$+x$
pz	$+y$	$+x$
nz	$+y$	$-x$

Next, these six views (which should be rendered using a square 90° field of view to completely cover all directions) are combined into a single environment map. This can be done from the RIB file:

`MakeCubeFaceEnvironment` *px nx py ny pz nz envfile*

Here *px*, *nx*, and so on are the names of the files containing the individual face images, and *envfile* is the name of the file where you would like the final environment map to be placed.

Alternatively, most renderers will have a separate program that will assemble an environment map from the six individual views. In the case of *PRMan*, this can be done with the `txmake` program:

`txmake -envcube` *px nx py ny pz nz envfile*

2.2.3 Flat Surface Reflection Maps

For the special case of flat objects (such as floors or flat mirrors), there is an even easier and more efficient method for producing reflections, which also solves the problem of environment maps being inaccurate for flat objects.

For the example of a flat mirror, we can observe that the image in the reflection would be identical to the image that you would get if you put another copy of the room on the other side of the mirror, *reflected* about the plane of the mirror. This geometric principle is illustrated in Figure 2.4.

Once we create this reflection map, we can turn it into a texture and index it from our shader. Because the pixels in the reflection map correspond exactly to the reflected image in the same pixels of the main image, we access the texture map by the texture's pixel coordinates, not the s, t coordinates of the mirror. We can do this by projecting `P` into `"NDC"` space. This is done in the `ReflMap` function in Listing 2.6.

2.2.4 General Reflections and Shiny Surfaces

We would prefer to write our shaders so that we may use either reflection or environment maps. Therefore, we can combine both into a single routine, `SampleEnvironment()`, given in List-

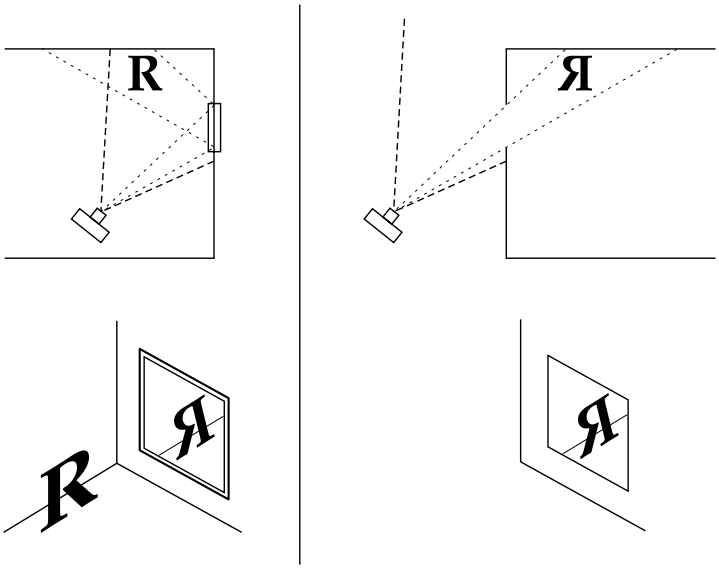


Figure 2.4: Creating a mirrored scene for generating a reflection map. On the top left, a camera views a scene that includes a mirror. Below is the image it produces. On the top right, the camera instead views the *mirror scene*. Notice that the image it produces (below) contains the required reflection.

ing 2.7.

A few comments on the source code in Listing 2.7. To allow easy specification of the many environment-related parameters, we define macros `ENVPARAMS`, `DECLARE_ENVPARAMS`, and `DECLARE_DEFAULTED_ENVPARAMS`, which are macros containing the parameter names, declarations, and declarations with default values, respectively. These macros allow us to succinctly include them in any shader, as we have done in shader `shinymetal` (Listing 2.8).

With this routine in our toolbox, we can make a straightforward shader that can be used for shiny metals like mirrors, chrome, or copper (see Listing 2.8). If you are using the shader for a flat object

Listing 2.6 `ReflMap` function performs simple reflection mapping.

```
color ReflMap ( string reflname;  point P; float blur;
                output float alpha; )
{
    /* Transform to the space of the environment map */
    point Pndc = transform ("NDC", P);
    float x = xcomp(Pndc), y = ycomp(Pndc);
    alpha = float texture (reflname[3], x, y, "blur", blur, "fill", 1);
    return color texture (reflname, x, y, "blur", blur);
}
```

Listing 2.7 `SampleEnvironment` function makes calls to either or both of `Environment` or `ReflMap` as needed.

```
#define ENVPARAMS     envname, envspace, envrad

#define DECLARE_ENVPARAMS                                   \
        string envname, envspace;  uniform float envrad

#define DECLARE_DEFAULTED_ENVPARAMS                         \
        string envname = "", envspace = "world";            \
        uniform float envrad = 100


color
SampleEnvironment (point P;  vector R;  float Kr, blur;  DECLARE_ENVPARAMS;)
{
    color C = 0;
    float alpha;
    if (envname != "") {
        if (envspace == "NDC")
             C = ReflMap (envname, P, blur, alpha);
        else C = Environment (envname, envspace, en-
vrad, P, R, blur, alpha);
    }
    return Kr*C;
}
```

with a prepared flat reflection map, you need only pass `"NDC"` as the `envspace` parameter and the `SampleEnvironment` function will correctly access your reflection map.

2.2.5 Fresnel and Shiny Plastic

All materials are more reflective at glancing angles than face-on (in fact, materials approach 100% reflectivity at grazing angles). For polished metals, the face-on reflectivity is so high that this difference in reflectivity with angle is not very noticeable. You can convince yourself of this by examining an ordinary mirror — it appears nearly equally shiny when you view your reflection head-on versus when you look at the mirror at an angle. So we tend to ignore the angular effect on reflectivity, assuming for simplicity that polished metals are equally reflective in all directions (as we have in the previous section).

Dielectrics such as plastic, ceramics, and glass are much less reflective than metals overall, and especially so when viewed face-on. Therefore, their higher reflectivity at grazing angles is a much more significant visual detail (see Figure 2.5). The formulas that relate angle to reflectivity of such materials are known as the *Fresnel equations*.

A function that calculates the Fresnel terms is included in Shading Language:

```
void fresnel (vector I; normal N; float eta;
              output float Kr, Kt; output vector R, T);
```

According to Snell's law and the Fresnel equations, `fresnel` computes the reflection and transmission direction vectors `R` and `T`, respectively, as well as the scaling factors for reflected and transmitted light, `Kr` and `Kt`. The `I` parameter is the normalized incident ray, `N` is the

Listing 2.8 `MaterialShinyMetal` and the `shinymetal` shader.

```
/* Compute the color of the surface using a simple metal-like BRDF.  To
 * give a metallic appearance, both diffuse and specular components are
 * scaled by the color of the metal.  It is recommended that Kd < 0.1,
 * Ks > 0.5, and roughness > 0.15 to give a believable metal-
lic appearance.
 */
color MaterialShinyMetal (normal Nf;  color basecolor;
                          float Ka, Kd, Ks, roughness, Kr, blur;
                          DECLARE_ENVPARAMS;)
{
    extern point P;
    extern vector I;
    vector IN = normalize(I), V = -IN;
    vector R = reflect (IN, Nf);
    return basecolor * (Ka*ambient() + Kd*diffuse(Nf) +
                        Ks*specular(Nf,V,roughness) +
                        SampleEnvironment (P, R, Kr, blur, ENVPARAMS));
}


surface
shinymetal ( float Ka = 1, Kd = 0.1, Ks = 1, roughness = 0.2;
             float Kr = 0.8, blur = 0;  DECLARE_DEFAULTED_ENVPARAMS; )
{
    normal Nf = faceforward (normalize(N), I);
    Ci = MaterialShinyMetal (Nf, Cs, Ka, Kd, Ks, roughness, Kr, blur,
                             ENVPARAMS);
    Oi = Os;  Ci *= Oi;
}
```

normalized surface normal, and `eta` is the ratio of refractive index of the medium containing `I` to that on the opposite side of the surface.

We can use `fresnel()` to attenuate the relative contributions of environmental and diffuse reflectances. This is demonstrated in the `MaterialShinyPlastic` and `shinyplastic`, both shown in Listing 2.9.

The index of refraction of air is very close to 1.0, water is 1.33, ordinary window glass is approximately 1.5, and most plastics are close to this value. In computer graphics, we typically assume that 1.5 is a reasonable refractive index for a very wide range of plastics and glasses. Therefore,

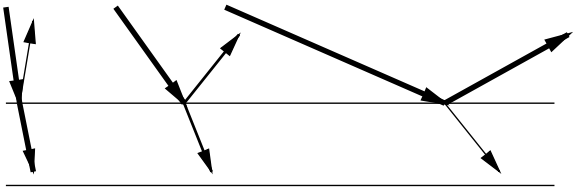


Figure 2.5: The ratio of reflected to transmitted light at a material boundary changes with the angle of the incident light ray.

a reasonable value for the `eta` parameter passed to `fresnel()` is 1/1.5. Nearly any optics textbook will list the indices of refraction for a variety of real materials, should you wish to be more physically accurate.

Of course, for real materials the refractive index is wavelength-dependent. We tend to ignore this effect, since we have such an ad hoc approach to spectral sampling and color spaces anyway. Although metals also have wavelength-dependent indices of refraction, fresnel effects, and angle-dependent spectral reflectivities, they are much less visually noticeable in most metals, so we usually just pretend that shiny metals reflect uniformly. You are, of course, perfectly free to go to the extra effort of computing the wavelength and angle dependency of metals!

Important note: In `MaterialShinyPlastic` we set `fkt=1-fkr`, overriding the value returned by `fresnel()`. The reasons for this are extremely subtle and well beyond the scope of this book. Suffice it to say that the value that `fresnel()` is supposed to return for `Kt` assumes a more rigorous simulation of light propagation than either *PRMan* or *BMRT* provides. In light of these inaccuracies, the intuitive notion that `Kt` and `Kr` ought to sum to 1 is about as good an approximation as one might hope for. We will therefore continue to use this oversimplification.

Listing 2.9 `shinyplastic` shader is for highly polished plastic and uses `fresnel()` so that reflections are stronger at grazing angles.

```
color
MaterialShinyPlastic (normal Nf; color basecolor;
                      float Ka, Kd, Ks, roughness, Kr, blur, ior;
                      DECLARE_ENVPARAMS; )
{
    extern vector I;   extern point P;
    vector IN = normalize(I), V = -IN;
    float fkr, fkt;  vector R, T;
    fresnel (IN, Nf, 1/ior, fkr, fkt, R, T);
    fkt = 1-fkr;
    return  fkt * basecolor * (Ka*ambient() + Kd*diffuse(Nf))
          + (Ks*fkr) * specular(Nf,V,roughness)
          + SampleEnvironment (P, R, fkr*Kr, blur, ENVPARAMS));
}


surface
shinyplastic ( float Ka = 1, Kd = 0.5, Ks = .5, roughness = 0.1;
               float Kr = 1, blur = 0, ior = 1.5;
               DECLARE_DEFAULTED_ENVPARAMS;)
{
    normal Nf = faceforward (normalize(N), I);
    Ci = MaterialShinyPlastic (Nf, Cs, Ka, Kd, Ks, rough-
ness, Kr, blur, ior,
                               ENVPARAMS);
    Oi = Os;  Ci *= Oi;
}
```

Figure 2.6: `MaterialShinyMetal` (left) and `MaterialShinyPlastic` (right).

2.3 Illuminance Loops, or How `diffuse()` and `specular()` Work

We have seen classes of materials that can be described by various relative weightings and uses of the built-in `diffuse()` and `specular()` functions. In point of fact, these are just examples of possible illumination models, and it is quite useful to write alternative ones. In order to do that, however, the shader needs access to the lights: how many are there, where are they, and how bright are they?

The key to such functionality is a special syntactic structure in Shading Language called an `illuminance` loop:

```
illuminance ( point position ) {
    statements;
}

illuminance ( point position; vector axis; float angle ) {
    statements;
}
```

The `illuminance` statement loops over all light sources visible from a particular *position*. In the first form, all lights are considered, and in the second form, only those lights whose directions are within *angle* of *axis* (typically, *angle*=$\pi/2$ and *axis*=`N`, which indicates that all light sources in the visible hemisphere from `P` should be considered). For each light source, the *statements* are executed, during which two additional variables are defined: `L` is the vector that points to the light source, and `Cl` is the color representing the incoming energy from that light source.

Perhaps the most straightforward example of the use of `illuminance` loops is the implementation of the `diffuse()` function:

```
color diffuse (normal Nn)
{
    extern point P;
    color C = 0;
    illuminance (P, Nn, PI/2) {
        C += Cl * (Nn . normalize(L));
    }
    return C;
}
```

Briefly, this function is looping over all light sources. Each light's contribution is computed using a Lambertian shading model; that is, the light reflected diffusely is the light arriving from the source multiplied by the dot product of the normal with the direction of the light. The contributions of all lights are summed and that sum is returned as the result of `diffuse()`.

2.4 Identifying Lights with Special Properties

In reality, surfaces simply scatter light. The scattering function, or BRDF (which stands for bidirectional reflection distribution function) may be quite complex. Dividing the scattering function into diffuse versus specular, or considering the BRDF to be a weighted sum of the two, is simply a convenient oversimplification. Many other simplifications and abstractions are possible.

In the physical world, light scattering is a property of the surface material, not a property of the light itself or of its source. Nonetheless, in computer graphics it is often convenient and desirable to place light sources whose purpose is solely to provide a highlight, or alternatively to provide a soft fill light where specular highlights would be undesirable. Therefore, we would like to construct our `diffuse()` function so that it can ignore light sources that have been tagged as being "nondiffuse"; that is, the source itself should only contribute specular highlights. We can do this by exploiting the *message passing* mechanism of Shading Language, wherein the surface shader may peek at the parameters of the light shader.

`float lightsource ( string paramname; output` *`type`* `result )`

> The `lightsource()` function, which may only be called from within an `illuminance` loop, searches for a parameter of the light source named `paramname`. If such a parameter is found and if its type matches that of the variable `result`, then its value will be stored in the `result` and the `lightsource()` function will return 1.0. If no such parameter is found, the variable `result` will be unchanged and the return value of `lightsource` will be zero.

We can use this mechanism as follows. Let us assume rather arbitrarily that any light that we wish to contribute only to specular highlights (and that therefore should be ignored by the `diffuse()` function) will contain in its parameter list an `output float` parameter named `__nondiffuse`. Similarly, we can use an `output float` parameter named `__nonspecular` to indicate that a particular light should not contribute to specular highlights. Then implementations of `diffuse()` and `specular()`, as shown in Listing 2.10, would respond properly to these controls. In both *PRMan* and *BMRT*, the implementations of `diffuse()` and `specular()` respond to `__nondiffuse` and `__nonspecular` in this manner (although as we explained earlier, the implementation of `specular()` is not quite what is in Listing 2.10).

Listing 2.10 The implementation of the built-in `diffuse()` and `specular()` functions, including controls to ignore nondiffuse and nonspecular lights.

```
color diffuse (normal Nn)
{
    extern point P;
    color C = 0;
    illuminance (P, Nn, PI/2) {
        float nondiff = 0;
        lightsource ("__nondiffuse", nondiff);
        C += Cl * (1-nondiff) * (Nn . normalize(L));
    }
    return C;
}


color specular (normal Nn; vector V; float roughness)
{
    extern point P;
    color C = 0;
    illuminance (P, Nn, PI/2) {
        float nonspec = 0;
        lightsource ("__nonspecular", nonspec);
        vector H = normalize (normalize(L) + V);
        C += Cl * (1-nonspec) * pow (max (0, Nn.H), 1/roughness);
    }
    return C;
}
```

This message passing mechanism may be used more generally to pass all sorts of "extra" information from the lights to the surfaces. For example, a light may include an `output` parameter giving its *ultraviolet* illumination, and special surface shaders may respond to this parameter by exhibiting fluorescence.

There is an additional means of controlling `illuminance` loops with an optional *light category* specifier:

```
illuminance ( string category; point position )
    statements;

illuminance ( string category; point position;
              vector axis; float angle )
    statements;
```

Ordinary `illuminance` loops will execute their body for every nonambient light source. The named category extension to the `illuminance` syntax causes the *statements* to be executed only for a subset of light sources.

Light shaders can specify the categories to which they belong by declaring a string parameter named `__category` (this name has two underscores), whose value is a comma-separated list of categories into which the light shader falls. When the `illuminance` statement contains a string parameter *category*, the loop will only consider lights for which the *category* is among those listed in

its comma-separated `__category` list. If the `illuminance` *category* begins with a `-` character, then only lights *not* containing that category will be considered. For example,

```
float uvcontrib = 0;
illuminance ("uvlight", P, Nf, PI/2) {
    float uv = 0;
    lightsource ("uv", uv);
    uvcontrib += uv;
}
Ci += uvcontrib * uvglowcolor;
```

will look specifically for lights containing the string `"uvlight"` in their `__category` list and will execute those lights, summing their `"uv"` output parameter. An example light shader that computes ultraviolet intensity might be

```
light
uvpointlight (float intensity = 1, uvintensity = 0.5;
              color lightcolor = 1;
              point from = point "shader" (0,0,0);
              output varying float uv = 0;
              string __category = "uvlight";)
{
    illuminate (from) {
        Cl = intensity * lightcolor / (L . L);
        uv = uvintensity / (L . L);
    }
}
```

Don't worry too much that you haven't seen light shaders yet: Before the end of this chapter, this example will be crystal clear.

2.5 Custom Material Descriptions

We have seen that the implementation of `diffuse()` is that of a Lambertian shading model that approximates a rough surface that scatters light equally in all directions. Similarly, `specular()` implements a Blinn-Phong scattering function (according to the RI spec). As we've seen, different weighted combinations of these two functions can yield materials that look like a variety of plastics and metals. Now that we understand how they operate, we may use the `illuminance` construct ourselves to create custom primitive local illumination models. Past ACM SIGGRAPH proceedings are a treasure-trove of ideas for more complex and realistic local illumination models. In this section we will examine three local illumination models (two physically based and one ad hoc) and construct shader functions that implement them.

2.5.1 Rough Surfaces

Lambert's law models a perfectly smooth surface that reflects light equally in all directions. This is very much an oversimplification of the behavior of materials. In the proceedings of SIGGRAPH '94, Michael Oren and Shree K. Nayar described a surface scattering model for rough surfaces. Their model (and others, including Beckman, Blinn, and Cook/Torrance) considers rough surfaces to have microscopic grooves and hills. This is modeled mathematically as a collection of *microfacets*

having a statistical distribution of relative directions. Their results indicated that many real-world rough materials (like clay) could be more accurately modeled using the following equation:

$$L_r(\theta_r, \theta_i, \phi_r - \phi_i, \sigma) = \frac{\rho}{\pi} E_0 \cos\theta_i (A + B \max[0, \cos(\phi_r - \phi_i)] \sin\alpha \tan\beta)$$

where

$$\begin{aligned} A &= 1.0 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33} \\ B &= 0.45\frac{\sigma^2}{\sigma^2 + 0.09} \\ \alpha &= \max[\theta_i, \theta_r] \\ \beta &= \min[\theta_i, \theta_r] \end{aligned}$$

and the terms mean

- ρ is the reflectivity of the surface (`Kd*Cs`).
- E_0 is the energy arriving at the surface from the light (`Cl`).
- θ_i is the angle between the surface normal and the direction of the light source.
- θ_r is the angle between the surface normal and the vector in the direction the light is reflected (i.e., toward the viewer).
- $\phi_r - \phi_i$ is the angle (about the normal) between the incoming and reflected light directions.
- σ is the standard deviation of the angle distribution of the microfacets (in radians). Larger values represent more rough surfaces; smaller values represent smoother surfaces. If $\sigma = 0$, the surface is perfectly smooth, and this function reduces to a simple Lambertian reflectance model. We'll call this parameter "roughness."

These equations are easily translated into an `illuminance` loop, as shown in Listing 2.11.

Figure 2.7 shows a teapot with the Oren/Nayar reflectance model. The left teapot uses a roughness coefficient of 0.5, while the right uses a roughness coefficient of 0, which has reflectance identical to Lambertian shading. Notice that as roughness increases, a strong back-scattering effect is present. The object begins to act as a retro-reflector, so that light that comes from the same direction as the viewer bounces back at a rate nearly independent of surface orientation.

It is this type of backscattering that accounts for the appearance of the full moon. Everyone has noticed that a full moon (when the viewer and sun are nearly at the same angle to the moon) looks like a flat disk, rather than like a ball. The moon is not Lambertian – it is more closely modeled as a rough surface, and this reflection model is a good approximation to the behavior of lunar dust.

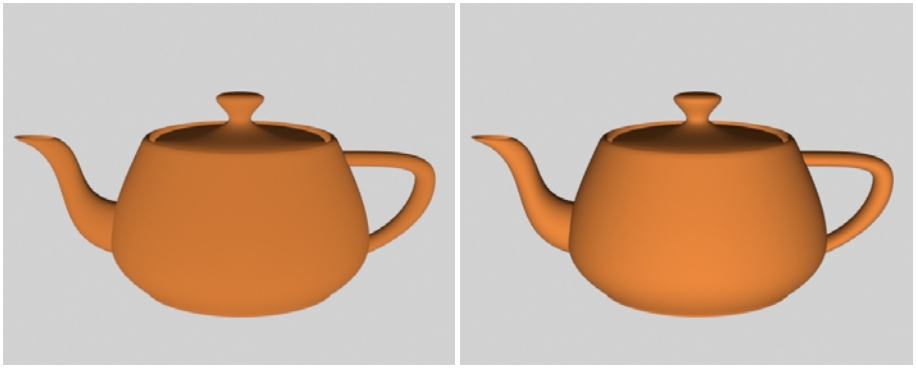
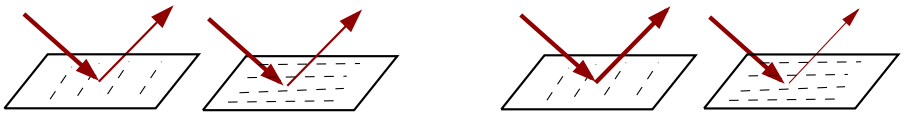


Figure 2.7: When the light source is directly behind the viewer, the Oren/Nayar model (left) acts much more as a retroreflector, compared to the Lambertian model (right).

2.5.2 Anisotropic Metal

The `MaterialRoughMetal` function described earlier does an adequate job of simulating the appearance of a metal object that is rough enough that coherent reflections (from ray tracing or environment maps) are not necessary. However, it does make the simplifying assumption that the metal scatters light only according to the the angular relationship between the surface normal, the eye, and the light source. It specifically does not depend on the orientation of the surface as it spins around the normal vector.

To help visualize the situation, consider the following diagram:



Imagine rotating the material around the normal vector. If the reflectivity in a particular direction is independent of the surface orientation, then the material is said to be *isotropic*. On the other hand, if the material reflects preferentially depending on surface orientation, then it is *anisotropic*.

Anisotropic materials are not uncommon. Various manufacturing processes can produce materials with microscopic grooves that are all aligned to a particular direction (picture the surface being covered with tiny half-cylinders oriented in parallel or otherwise coherently). This gives rise to anisotropic BRDFs. A number of papers have been written about anisotropic reflection models, including (Kajiya, 1985; Poulin and Fournier, 1990).

Greg Ward Larson described an anisotropic reflection model in his SIGGRAPH '92 paper, "Measuring and Modeling Anisotropic Reflection." (Ward, 1992). In this paper, anisotropic specular reflection was given as:

$$\frac{1}{\sqrt{\cos\theta_i \cos\theta_r}} \frac{1}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{\hat{h}\cdot\hat{x}}{\alpha_x}\right)^2 + \left(\frac{\hat{h}\cdot\hat{y}}{\alpha_y}\right)^2}{1+\hat{h}\cdot\hat{n}}\right]$$

where

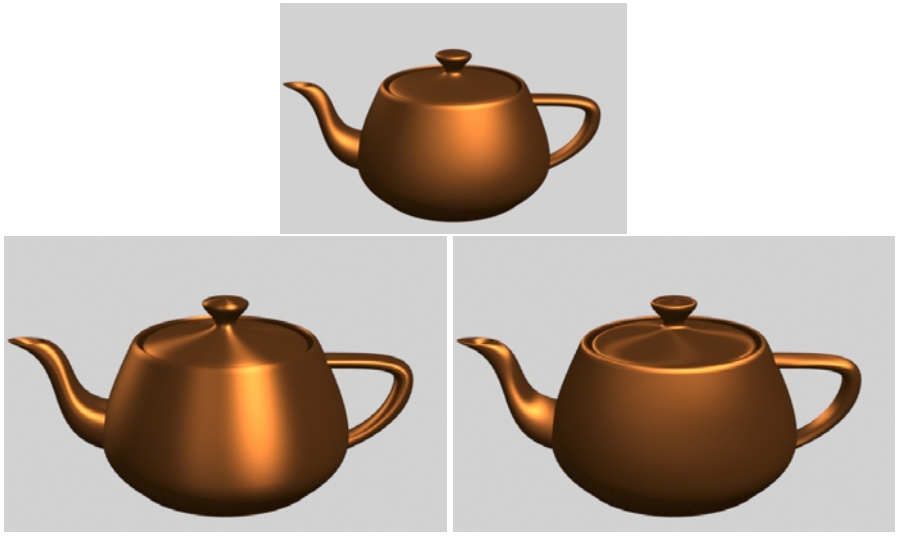


Figure 2.8: Examples of the Ward anisotropic reflection model. Isotropic reflection with `xroughness`=`yroughness`=0.3 (top). Anisotropic reflection with `xroughness`=0.15, `yroughness`=0.5 (bottom left). Anisotropic reflection with `xroughness`=0.5, `yroughness`=0.15 (bottom right). In all cases, `xdir`=`normalize(dPdu)`.

- θ_i is the angle between the surface normal and the direction of the light source.
- θ_r is the angle between the surface normal and the vector in the direction the light is reflected (i.e., toward the viewer).
- $\hat{x}$ and $\hat{y}$ are the two perpendicular tangent directions on the surface.
- α_x and α_y are the standard deviations of the slope in the $\hat{x}$ and $\hat{y}$ directions, respectively. We will call these `xroughness` and `yroughness`.
- $\hat{n}$ is the unit surface normal (`normalize(N)`).
- $\hat{h}$ is the half-angle between the incident and reflection rays (i.e., `H` = `normalize (normalize(-I) + normalize(L))`).

Listing 2.12 lists the function `LocIllumWardAnisotropic`, which implements the anisotropic specular component of Larson's model.[3] This function can be used instead of an ordinary `specular()` call. It differs from `specular()` in that it takes *two* roughness values: one for the direction aligned with surface tangent `xdir`, and the other for the perpendicular direction. Figure 2.8 shows this model applied to a teapot.

[3]When comparing the original equation to our SL implementation, you may wonder where the factor of $1/\pi$ went, and why there appears to be an extra factor of $L \cdot N$. This is not an error! Greg Ward Larson's paper describes the BRDF, which is only part of the kernel of the light integral, whereas shaders describe the result of that integral. This is something that must be kept in mind when coding traditional BRDFs in `illuminance` loops.

2.5.3 Glossy Specular Highlights

The previous subsections listed Shading Language implementations of two local illumination models that are physically based simulations of the way light reflects off certain material types. Many times, however, we want an effect that achieves a particular look without resorting to simulation. The look itself may or may not match a real-world material, but the computations are completely ad hoc. This subsection presents a local illumination model that achieves a useful look, but that is decidedly not based on the actual behavior of light. It works well for glossy materials, such as finished ceramics, glass, or wet materials.

We note that specular highlights are a lot like mirror reflections of the light source. The observation that they are big fuzzy circles on rough objects and small sharp circles on smooth objects is a combination of the blurry reflection model and the fact that the real light sources are not infinitesimal points as we often use in compute graphics, but extended area sources such as light bulbs, the sun, and so on.

For polished glossy surfaces, we would like to get a clear, distinct reflection of those bright area sources, even if there isn't a noticeable mirror reflection of the rest of the environment (and even if we aren't really using true area lights). We propose that we could achieve this look by thresholding the specular highlight. In other words, we modify a Blinn-Phong specular term (c.f. Shader Listing 2.10) from

```
Cl * pow (max (0, Nn.H), 1/roughness);
```

to the thresholded version:

```
Cl * smoothstep (e0, e1, pow (max (0, Nn.H), 1/roughness));
```

With an appropriately chosen `e0` and `e1`, the specular highlight will appear to be smaller, have a sharp transition, and be fully bright inside the transition region. Shader Listing 2.13 is a full implementation of this proposal (with some magic constants chosen empirically by the authors). Finally, Figure 2.9 compares this glossy specular highlight to a standard plastic-like `specular()` function. In that example, we used `roughness`=0.1, `sharpness`=0.5.

2.6 Light Sources

Previous sections discuss how surface shaders respond to the light energy that arrives at the surface, reflecting in different ways to give the appearance of different types of materials. Now it is time to move to light shaders, which allow the shader author similarly detailed control over the operation of the light sources themselves.

2.6.1 Non-Geometric Light Shaders

Light source shaders are syntactically similar to surface shaders. The primary difference is that the shader type is called `light` rather than `surface` and that a somewhat different set of built-in variables is available. The variables available inside light shaders are listed in Table 2.2. The goal of a light shader is primarily to determine the radiant energy `Cl` and light direction `L` of light impinging on `Ps` from this source. In addition, the light may compute additional quantities and store

Figure 2.9: Comparing the glossy versus plastic specular illumination models.

Table 2.2: Global variables available inside light shaders. Variables are read-only except where noted.

`point Ps`	Position of the point *on the surface* that requested data from the light shader.
`vector L`	The vector giving the direction of outgoing light from the source to the point being shaded, `Ps`. This variable can be set explicitly by the shader but is generally set implicitly by the `illuminate` or `solar` statements.
`color Cl`	The light color of the energy being emitted by the source. Setting this variable is the primary purpose of a light shader.

them in its `output` variables, which can be read and acted upon by `illuminance` loops using the `lightsource` statement.

The most basic type of light is an ambient source, which responds in a way that is independent of position. Such a shader is shown in Shader Listing 2.14. The `ambientlight` shader simply sets `Cl` to a constant value, determined by its parameters. It also explicitly sets `L` to zero, indicating to the renderer that there is no directionality to the light.

For directional light, although we could explicitly set `L`, there are some syntactic structures for emitting light in light shaders that help us do the job efficiently. One such syntactic construct is the `solar` statement:

```
solar ( vector axis; float spreadangle ) {
    statements;
}
```

The effect of the `solar` statement is to send light to every `Ps` from the same direction, given by *axis*. The result is that rays from such a light are parallel, as if the source was infinitely far away. An example of such a source would be the sun.

Listing 2.15 is an example of the `solar` statement. The `solar` statement implicitly sets the `L` variable to its first argument; there is no need to set L yourself. Furthermore, the `solar` statement will compare this direction to the light gathering cone of the corresponding `illuminance` statement in the surface shader. If the light's `L` direction is outside the range of angles that the `illuminance` statement is gathering, the block of statements within the `solar` construct will not be executed.

The *spreadangle* parameter is usually set to zero, indicating that the source subtends an infinitesimal angle and that the rays are truly parallel. Values for *spreadangle* greater than zero indicate that a plethora of light rays arrive at each `Ps` from a range of directions, instead of a single ray from a particular direction. Such lights are known as *broad solar lights* and are analogous to very distant but very large area lights (for example, the sun actually subtends a 1/2 degree angle when seen from Earth). The exact mechanism for handling these cases may be implementation-dependent, differing from renderer to renderer.

For lights that have a definite, finitely close position, there is another construct to use:

```
illuminate ( point from ) {
    statements;
}
```

This form of the `illuminate` statement indicates that light is emitted from position *from*, and is radiated in all directions. As before, `illuminate` implicitly sets `L = Ps -` *from*. Listing 2.16 shows an example of a simple light shader that radiates light in all directions from a point `from` (which defaults to the origin of light shader space[4]). Dividing the intensity by `L.L` (which is the square of the length of `L`) results in what is known as $1/r^2$ falloff. In other words, the energy of light impinging on a surface falls off with the square of the distance between the surface and the light source.

A second form of `illuminate` also specifies a particular cone of light emission, given by an axis and angle:

```
illuminate ( point from; vector axis; float angle ) {
    statements;
}
```

An example use of this construct can be found in the standard `spotlight` shader, shown in Listing 2.17. The `illuminate` construct will prevent light from being emitted in directions outside the cone. In addition, the shader computes $1/r^2$ distance falloff, applies a cosine falloff to directions away from the central axis, and smoothly fades the light out at the edges of the cone.

Both forms of `illuminate` will check the corresponding `illuminance` statement from the surface shader, which specified a cone axis and angle from which to gather light. If the *from* position is outside this angle range, the body of statements inside the `illuminate` construct will be skipped, thus saving computation for lights whose results would be ignored.

[4] In a light shader, `"shader"` space is the coordinate system that was in effect at the point that the `LightSource` statement appeared in the RIB file.

The lights presented here are very simple examples of light shaders, only meant to illustrate the `solar` and `illuminate` constructs. For high-quality image generation, many more controls would be necessary, including more flexible controls over the light's cross-sectional shape, directional and distance falloff, and so on. Such controls are discussed in detail in Chapter 14 of *Advanced RenderMan: Creating CGI for Motion Pictures* (Lighting Controls for Computer Cinematography).

2.6.2 Area Light Sources

Area light sources are those that are associated with geometry. Table 2.3 lists the variables available inside area light sources. Many of the variables available to the area light shader describe a position *on the light* that has been selected by the renderer as the point at which the light shader is sampled. You need not worry about how this position is selected — the renderer will do it for you.

Table 2.3: Global variables available inside area light shaders. Variables are read-only except where noted.

`point Ps`	Position of the point *on the surface* that requested data from the light shader.
`point P`	Position of the point *on the light*.
`normal N`	The surface normal of the light source geometry (at `P`).
`float u, v, s, t`	The 2D parametric coordinates of `P` (on the light source geometry).
`vector dPdu vector dPdv`	The partial derivatives (i.e., tangents) of the light source geometry at `P`.
`vector L`	The vector giving the direction of outgoing light from the source to the point being shaded, `Ps`. This variable can be set explicitly by the shader but is generally set implicitly by the `illuminate` or `solar` statements.
`color Cl`	The light color of the energy being emitted by the source. Setting this variable is the primary purpose of a light shader.

Positions, normals, and parameters on the light only make sense if the light is an area light source defined by a geometric primitive. If the light is an ordinary point source rather than an area light, these variables are undefined. Note that *PRMan* does not support area lights. Therefore, in that renderer the light geometry variables are undefined and should not be trusted to contain any meaningful data.

An example light shader that could be used for a simple area light source is given in Listing 2.18. Note the similarity to the `pointlight` shader — the main difference is that rather than using a `from` parameter as the light position, we illuminate from the position `P` that the renderer chose for us by sampling the area light geometry. This shader illuminates only points on the *outside* of the light source geometry. It would also be fine to simply use `pointlight`, if you wanted the area light geometry to illuminate in all directions.

2.6.3 Shadows

Notice that light shaders are strictly "do-it-yourself" projects. If you want color, you have to specify it. If you want falloff, you need to code it (in the case of `distantlight` we have no distance-based falloff; for `spotlight` and `pointlight` we used $1/r^2$ falloff). Similarly, if you want the lights to be shadowed, that also needs to be in the shader.

A number of shadowing algorithms have been developed over the years, and their relative merits depend greatly on the overall rendering architectures. So as not to make undue demands on the renderer, the RenderMan standard provides for the "lowest common denominator": shadow maps. Shadow maps are simple, relatively cheap, very flexible, and can work with just about any rendering architecture.

The shadow map algorithm works in the following manner. Before rendering the main image, we will render separate images *from the vantage points of the lights*. Rather than render RGB color images, these light source views will record depth only (hence the name, *depth map*). An example depth map can be seen in Figure 2.10.

Once these depth maps have been created, we can render the main image from the point of view of the camera. In this pass, the light shader can determine if a particular surface point is in shadow by comparing its distance to the light against that stored in the shadow map. If it matches the depth in the shadow map, it is the closest surface to the light in that direction, so the object receives light. If the point in question is *farther* than indicated by the shadow map, it indicates that some other object was closer to the light when the shadow map was created. In such a case, the point in question is known to be in shadow. Figure 2.10 shows a simple scene with and without shadows, as well as the depth map that was used to produce the shadows.

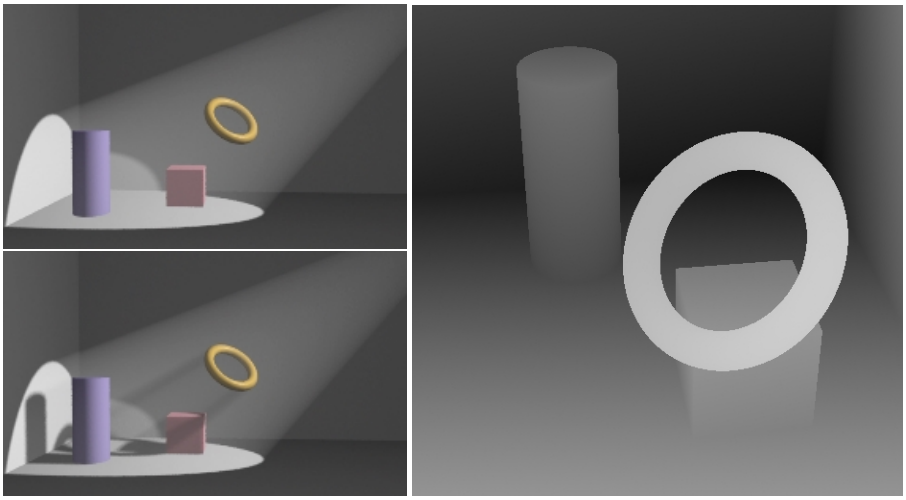


Figure 2.10: Shadow depth maps. A simple scene with and without shadows (left). The shadow map is just a depth image rendered from the point of view of the light source (right). To visualize the map, we assign white to near depths, black to far depths.

Shading Language gives us a handy built-in function to access shadow maps:

```
float shadow ( string shadowmapname; point Ptest; ...  )
```

The `shadow()` function tests the point `Ptest` (in `"current"` space) against the shadow map file specified by `shadowmapname`. The return value is 0.0 if `Ptest` is unoccluded, and 1.0 if `Ptest` is occluded (in shadow according to the map). The return value may also be between 0 and 1, indicating that the point is in partial shadow (this is very handy for soft shadows).

Like `texture()` and `environment()`, the `shadow()` call has several optional arguments that can be specified as token/value pairs:

- `"blur"` takes a `float` and controls the amount of blurring at the shadow edges, as if to simulate the penumbra resulting from an area light source (see Figure 2.11). A value of `"blur=0"` makes perfectly sharp shadows; larger values blur the edges. It is strongly advised to add some blur, as perfectly sharp shadows look unnatural and can also reveal the limited resolution of the shadow map.
- `"samples"` is a `float` specifying the number of samples used to test the shadow map. Shadow maps are antialiased by supersampling, so although having larger numbers of samples is more expensive, they can reduce the graininess in the blurry regions. We recommend a minimum of 16 samples, and for blurry shadows it may be quite reasonable to use 64 samples or more.
- `"bias"` is a `float` that *shifts the apparent depth of the objects from the light*. The shadow map is just an approximation, and often not a very good one. Because of numerical imprecisions in the rendering process and the limited resolution of the shadow map, it is possible for the shadow map lookups to incorrectly indicate that a surface is in partial shadow, even if the object is indeed the closest to the light. The solution we use is to add a "fudge factor" to the lookup to make sure that objects are pushed out of their own shadows. Selecting an appropriate bias value can be tricky. Figure 2.12 shows what can go wrong if you select a value that is either too small or too large.
- `"width"` is a `float` that multiplies the estimates of the rate of change of `Ptest` (used for antialiasing the shadow map lookup). This parameter functions analogously to the `"width"` parameter to `texture()` or `environment()`. Its use is largely obsolete and we recommend using `"blur"` to make soft shadow edges rather than `"width"`.

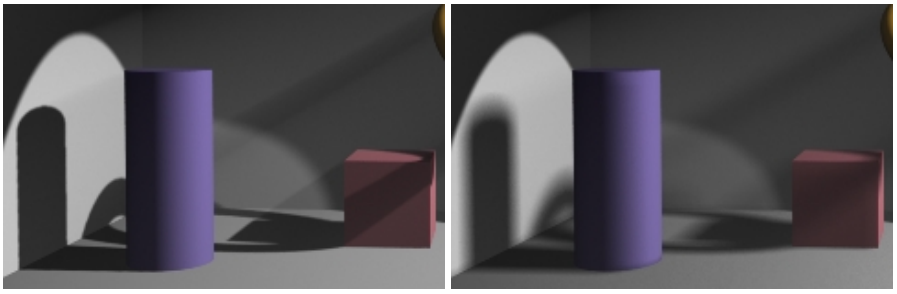


Figure 2.11: Adding blur to shadow map lookups can give a penumbra effect.

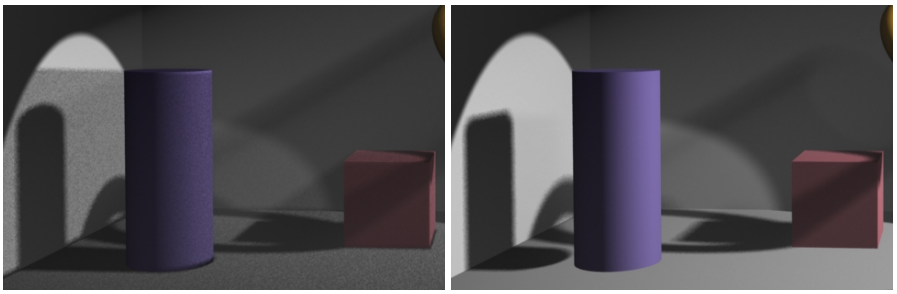


Figure 2.12: Selecting shadow bias. Too small a bias value will result in incorrect self-shadowing (left). Notice the darker, dirtier look compared to Figures 2.11 or 2.10. Too much bias can also introduce artifacts, such as the appearance of "floating objects" or the detached shadow at the bottom of the cylinder (right).

The `Ptest` parameter determines the point at which to determine how much light is shadowed, but how does the renderer know the point of origin of the light? When the renderer creates a shadow map, it also stores in the shadow file the origin of the camera at the time that the shadow map was made — in other words, the emitting point. The `shadow()` function knows to look for this information in the shadow map file. Notice that since the shadow origin comes from the shadow map file rather than the light shader, it's permissible (and often useful) for the shadows to be cast from an entirely different position than the point from which the light shader illuminates. Listing 2.19 shows a modification of the `spotlight` shader that uses a shadow map. This light shader is still pretty simple, but the entirety of Chapter 14 of *Advanced RenderMan: Creating CGI for Motion Pictures* discusses more exotic features in light shaders.

Here are some tips to keep in mind when rendering shadow maps:

- Select an appropriate shadow map resolution. It's not uncommon to use 2k $\times$ 2k or even higher-resolution shadow maps for film work.
- View the scene through the "shadow camera" before making the map. Make sure that the field of view is as small as possible, so as to maximize the effective resolution of the objects in the shadow map. Try to avoid your objects being small in the shadow map frame, surrounded by lots of empty unused pixels.
- Remember that depth maps must be one unjittered depth sample per pixel. In other words, the RIB file for the shadow map rendering ought to contain the following options:

```
PixelSamples 1 1
PixelFilter "box" 1 1
Hider "hidden" "jitter" [0]
Display "shadow.z" "zfile" "z"
ShadingRate 4
```

- In shadow maps, only depth is needed, not color. To save time rendering shadow maps, remove all `Surface` calls and increase the number given for `ShadingRate` (for example,

as above). If you have surface shaders that displace significantly and those bumps need to self-shadow, you may be forced to run the surface shaders anyway (though you can still remove the lights). Beware!

- When rendering the shadow map, only include objects that will actually cast shadows on themselves or other objects. Objects that only receive, but do not cast, shadows (such as walls or floors) can be eliminated from the shadow map pass entirely. This saves rendering time when creating the shadow map and also eliminates the possibility that poorly chosen bias will cause these objects to incorrectly self-shadow (since they aren't in the maps anyway).

- Some renderers may create shadow map files directly. Others may create only "depth maps" (or "z files") that require an additional step to transform them into full-fledged shadow maps (much as an extra step is often required to turn ordinary image files into texture maps). For example, when using *PRMan*, z files must be converted into shadow maps as follows:

```
txmake -shadow shadow.z shadow.sm
```

  This command invokes the `txmake` program (*PRMan*'s texture conversion utility) to read the raw depth map file `shadow.z` and write the shadowmap file `shadow.sm`.

It is also possible that some renderers (including *BMRT*, but not *PRMan*) support automatic ray-cast shadows which do not require shadow maps at all. In the case of *BMRT*, the following RIB attribute causes subsequently declared `LightSource` and `AreaLightSource` lights to automatically be shadowed:

```
Attribute "light" "shadows" ["on"]
```

There are also controls that let you specify which geometric objects cast shadows (consult the *BMRT* User's Manual for details). Chapter 17 of *Advanced RenderMan: Creating CGI for Motion Pictures* (Ray Tracing in PRMan), also discusses extensions to Shading Language that allow for ray-cast shadow checks in light shaders.

Further Reading

Early simple local illumination models for computer graphics used Lambertian reflectance. Bui Tuong Phong (Phong, 1975) proposed using a specular illumination model $(L \cdot R)^n$ and also noted that the appearance of faceted objects could be improved by interpolating the vertex normals. Phong's reflection model is still commonly used in simple renderers, particularly those implemented in hardware. Blinn reformulated this model as $(N \cdot H)^n$, with H defined as the angle halfway between L and N. This gives superior results, but for some reason few renderers or graphics boards bother to use this improved version.

The fundamentals of environment mapping can be found in (Greene, 1986a) and (Greene, 1986b).

The anisotropic specular illumination model that we use came from (Ward, 1992). The reader is directed to that work for more information on the derivation, details, and use of Greg Ward Larson's model. Additional anisotropic local illumination models can be found in (Kajiya, 1985), and (Poulin and Fournier, 1990). Oren and Nayar's generalization of Lambert's law can be found in (Oren and Nayar, 1994). A similar reflection model for simulation of clouds, dusty, and rough surfaces can

be found in (Blinn, 1982). Treatments of iridescence can be found in (Smits and Meyer, 1989) and (Gondek et al., 1994).

An excellent overall discussion of surface physics, including refraction and the Fresnel equations (derived in all their gory detail) can be found in (Hall, 1989). This book contains equations and pseudocode for many of the more popular local illumination models. Unfortunately, it has come to our attention that this book is now out of print. Glassner's book (Glassner, 1995) also is an excellent reference on BRDFs.

Additional papers discussing local illumination models include (Blinn and Newell, 1976), (Blinn, 1977), (Cook and Torrance, 1981), (Whitted and Cook, 1985; Whitted and Cook, 1988), (Hall, 1986), (Nakamae et al., 1990), (He et al., 1991), (Westin et al., 1992), (Schlick, 1993), (Hanrahan and Krueger, 1993), (Lafortune et al., 1997), (Goldman, 1997).

Shadow maps are discussed in (Williams, 1978) and (Reeves et al., 1987).

Listing 2.11 `LocIllumOrenNayar` implements a BRDF for diffuse, but rough, surfaces.

```
/*
 * Oren and Nayar's generalization of Lambert's reflection model.
 * The roughness parameter gives the standard deviation of angle
 * orientations of the presumed surface grooves.  When roughness=0,
 * the model is identical to Lambertian reflection.
 */
color
LocIllumOrenNayar (normal N;  vector V;  float roughness;)
{
    /* Surface roughness coefficients for Oren/Nayar's formula */
    float sigma2 = roughness * roughness;
    float A = 1 - 0.5 * sigma2 / (sigma2 + 0.33);
    float B = 0.45 * sigma2 / (sigma2 + 0.09);
    /* Useful precomputed quantities */
    float  theta_r = acos (V . N);        /* Angle between V and N */
    vector V_perp_N = normalize(V-N*(V.N)); /* Part of V perpendicu-
lar to N */

    /* Accumulate incoming radiance from lights in C */
    color  C = 0;
    extern point P;
    illuminance (P, N, PI/2) {
        /* Must declare extern L & Cl because we're in a function */
        extern vector L;  extern color Cl;
        float nondiff = 0;
        lightsource ("__nondiffuse", nondiff);
        if (nondiff < 1) {
            vector LN = normalize(L);
            float cos_theta_i = LN . N;
            float cos_phi_diff = V_perp_N . normalize(LN -
N*cos_theta_i);
            float theta_i = acos (cos_theta_i);
            float alpha = max (theta_i, theta_r);
            float beta = min (theta_i, theta_r);
            C += (1-nondiff) * Cl * cos_theta_i *
                (A + B * max(0,cos_phi_diff) * sin(alpha) * tan(beta));
        }
    }
    return C;

}
```

Listing 2.12 `LocIllumWardAnisotropic`: Greg Ward Larson's anisotropic specular illumination model.

```
/*
 * Greg Ward Larson's anisotropic specular local illumination model.
 * The derivation and formulae can be found in:  Ward, Gregory J.
 * "Measuring and Modeling Anisotropic Reflection," ACM Computer
 * Graphics 26(2) (Proceedings of Siggraph '92), pp. 265-272, July, 1992.
 * Notice that compared to the paper, the implementation below appears
 * to be missing a factor of 1/pi, and to have an extra L.N term.
 * This is not an error!  It is because the paper's formula is for the
 * BRDF, which is only part of the kernel of the light integral, whereas
 * shaders must compute the result of the integral.
 *
 * Inputs:
 *   N - unit surface normal
 *   V - unit viewing direction (from P toward the camera)
 *   xdir - a unit tangent of the surface defining the reference
 *          direction for the anisotropy.
 *   xroughness - the apparent roughness of the surface in xdir.
 *   yroughness - the roughness for the direction of the surface
 *          tangent perpendicular to xdir.
 */
color
LocIllumWardAnisotropic (normal N;  vector V;
                         vector xdir;  float xroughness, yroughness;)
{
    float sqr (float x) { return x*x; }

    float cos_theta_r = clamp (N.V, 0.0001, 1);
    vector X = xdir / xroughness;
    vector Y = (N ^ xdir) / yroughness;

    color C = 0;
    extern point P;
    illuminance (P, N, PI/2) {
        /* Must declare extern L & Cl because we're in a function */
        extern vector L;  extern color Cl;
        float nonspec = 0;
        lightsource ("__nonspecular", nonspec);
        if (nonspec < 1) {
            vector LN = normalize (L);
            float cos_theta_i = LN . N;
            if (cos_theta_i > 0.0) {
                vector H = normalize (V + LN);
                float rho = exp (-2 * (sqr(X.H) + sqr(Y.H)) / (1 + H.N))
                    / sqrt (cos_theta_i * cos_theta_r);
                C += Cl * ((1-nonspec) * cos_theta_i * rho);
            }
        }
    }
    return C / (4 * xroughness * yroughness);
}
```

Listing 2.13 `LocIllumGlossy` function: a nonphysical replacement for `specular()` that makes a uniformly bright specular highlight.

```
/*
 * LocIllumGlossy - a possible replacement for specular(), having
 * more uniformly bright core and a sharper falloff.  It's a nice
 * specular function to use for something made of glass or liquid.
 * Inputs:
 *  roughness - related to the size of the highlight, larger is bigger
 *  sharpness - 1 is infinitely sharp, 0 is very dull
 */
color LocIllumGlossy ( normal N;  vector V;
                       float roughness, sharpness; )
{
    color C = 0;
    float w = .18 * (1-sharpness);
    extern point P;
    illuminance (P, N, PI/2) {
        /* Must declare extern L & Cl because we're in a function */
        extern vector L;  extern color Cl;
        float nonspec = 0;
        lightsource ("__nonspecular", nonspec);
        if (nonspec < 1) {
            vector H = normalize(normalize(L)+V);
            C += Cl * ((1-nonspec) *
                       smoothstep (.72-w, .72+w,
                                   pow(max(0,N.H), 1/roughness)));
        }
    }
    return C;
}
```

Listing 2.14 Ambient light source shader.

```
light
ambientlight (float intensity = 1;
              color lightcolor = 1;)
{
    Cl = intensity * lightcolor;  /* doesn't depend on position */
    L = 0;                        /* no light direction */
}
```

Listing 2.15 `distantlight`, a light shader for infinitely distant light sources with parallel rays.

```
light
distantlight ( float intensity = 1;
               color lightcolor = 1;
               point from = point "shader" (0,0,0);
               point to = point "shader" (0,0,1); )
{
    solar (to-from, 0) {
        Cl = intensity * lightcolor;
    }
}
```

Listing 2.16 `pointlight` radiates light in all directions from a particular point.

```
light
pointlight (float intensity = 1;
            color lightcolor = 1;
            point from = point "shader" (0,0,0);)
{
    illuminate (from) {
        Cl = intensity * lightcolor / (L . L);
    }
}
```

Listing 2.17 `spotlight` radiates a cone of light in a particular direction.

```
light
spotlight ( float intensity = 1;
            color lightcolor = 1;
            point from = point "shader" (0,0,0);
            point to = point "shader" (0,0,1);
            float coneangle = radians(30);
            float conedeltaangle = radians(5);
            float beamdistribution = 2; )
{
    uniform vector A = normalize(to-from);
    uniform float cosoutside = cos (coneangle);
    uniform float cosinside  = cos (coneangle-conedeltaangle);
    illuminate (from, A, coneangle) {
        float cosangle = (L . A) / length(L);
        float atten = pow (cosangle, beamdistribution) / (L . L);
        atten *= smoothstep (cosoutside, cosinside, cosangle);
        Cl = atten * intensity * lightcolor;
    }
}
```

Listing 2.18 `arealight` is a simple area light shader.

```
light
arealight (float intensity = 1;
           color lightcolor = 1;)
{
    illuminate (P, N, PI/2) {
        Cl = (intensity / (L.L)) * lightcolor;
    }
}
```

Listing 2.19 `shadowspot` is just like spotlight, but casts shadows using a shadow depth map.

```
light
shadowspot ( float  intensity = 1;
             color  lightcolor = 1;
             point  from = point "shader" (0,0,0);
             point  to = point "shader" (0,0,1);
             float  coneangle = radians(30);
             float  conedeltaangle = radians(5);
             float  beamdistribution = 2;
             string shadowname = "";
             float  samples = 16;
             float  blur = 0.01;
             float  bias = 0.01; )
{
    uniform vector A = normalize(to-from);
    uniform float cosoutside = cos (coneangle);
    uniform float cosinside  = cos (coneangle-conedeltaangle);

    illuminate (from, A, coneangle) {
        float cosangle = (L . A) / length(L);
        float atten = pow (cosangle, beamdistribution) / (L . L);
        atten *= smoothstep (cosoutside, cosinside, cosangle);
        if (shadowname != "") {
            atten *= 1 - shadow (shadowname, Ps, "samples", samples,
                                 "blur", blur, "bias", bias);
        }
        Cl = atten * intensity * lightcolor;
    }
}
```

Chapter 3
Rendering Related Issues on the Production of Disney's *Dinosaur*

Tal Lancaster
Walt Disney Feature Animation
tlan@fa.disney.com

Abstract

This paper is broken into three clean sections:

1. Zero shadow bias
2. Camera Projections
3. Modeling with RenderMan

There were plenty of other topics to choose from. Like, encapsulating a particle and blobby system inside a surface shader; creating a 50 foot high concussion wave on an already turbulent ocean; using just a flat plane for geometry; ripping flesh off of characters (and having them heal up) with the aid of displacement shaders; cycling texture-maps for herd characters; CG grass; CG fur; and tons of other stuff. But as I only have one hour to present (rather than the eight that would be needed), I have picked these three to give a taste of what we did. Also, I hope that these are generally applicable and will inspire you to new directions.

So without further ado...

3.1 Zero shadow bias

It is common practice to post process the color images that one gets out of RenderMan. People come up all kinds of reasons to manipulate their images. However, you never hear of people mucking around with their shadow maps. It seems as though most people blissfully leave their shadow maps alone, but are eager to find ways to process, their color pass.

Now consider the possibilities of processing shadow maps. This section illustrates a technique we used on Dinosaur that entailed processing our shadow maps before using them in the color pass render. The result was that our lighters rarely needed to be concerned with setting the shadow-bias for any of their lights.

Beginning with *PhotoRealistic RenderMan* (*PRMan*) 3.8, there are some choices on what data the shadow maps should contain. The current choices for pixel depth values are: minimum, maximum, average, and midpoint.[1]

Before all these choices (pre-*PRMan* 3.8 days), shadow maps contained only minimum depth values. One of the biggest problems with the minimum depth maps was with self-shadowing. For numerous reasons (numerical inaccuracies being the main culprit), incorrect shadows would get produced. So minimum depth maps should be considered approximations. To help alleviate this problem, *PRMan* has shadow-bias controls. When reading the shadow-maps, these controls help to create a fudge factor that pushes the depth values back away from the front of the surface.

It is important to find the right fudge factor to apply. Because a shadow-bias that is too small will cause the surface to look dirty, due to inaccurate self-shadowing (see Figure 3.1, top). This is due to round-off error. A surface, in the shadow map, alternates believing that it is in and out of shadow because it is approximately represented in the map (instead of exactly). One also has to take care that the shadow-bias isn't too large or the shadow will get pushed back so much that it detaches from the object entirely (see Figure 3.1(bottom right)).[2]

Figure 3.1 shows the results with various shadow bias settings. In In the top two images, the bias is too small and so the images appear dirty. In the bottom right image, the bias is so large that the shadow has detached from the cylinder entirely. In the bottom left image, the bias is about right.

Frequently, the ideal shadow should read about midpoint of the object from the view of the light casting the shadow. This is what the goal should be most of the time to get one's shadows to read correctly. The *PRMan* "midpoint" method does precisely this, which is why one very rarely needs to worry about setting the shadow-bias when using this method.

Finding just the right shadow-bias is more of an art than science. There are many factors to consider. Large-scale objects typically will need larger shadow-bias values than smaller objects. Also, the shape of the object and its orientation to the light will effect the shadow-bias settings needed to get the shadow to look better. If the object or light change their orientations too much relative to each other, the bias, too will need to be changed accordingly (aka. the shadow-bias may need animating). Typically the actual shadow-bias number is determined by a trial-and-error method. Since, the object's orientation to the camera could effect the shadow-bias settings, each shadow light in the scene may need to have its shadow-bias values changed differently.

On Dinosaur we were locked off on *PRMan* 3.7. We never had access to any of the newer shadow generation options, like "midpoint", which minimizes the need for shadow-bias tweaking. So shadow-bias tweaking was a fact of life on our production.

Laurent Charbonnel, a Look-Development TD on Dinosaur, came up with a simple but ingenious method for allowing the lighters to ignore the shadow-bias much of the time. This technique was called "Zero Shadow Bias", as much of the time one could leave the shadow-bias setting at 0. Zero Shadow Bias is a good example of adding more intelligence to the shadow generation process.

[1] See *PRMan* 3.8 release notes:
`http://www.pixar.com/products/rendermandocs/toolkit/Toolkit/rnotes-3.8.html#shadowmap` for more information.

[2] See `http://www.pixar.com/products/rendermandocs/toolkit/Toolkit/AppNotes/appnote.15.html` for more information.

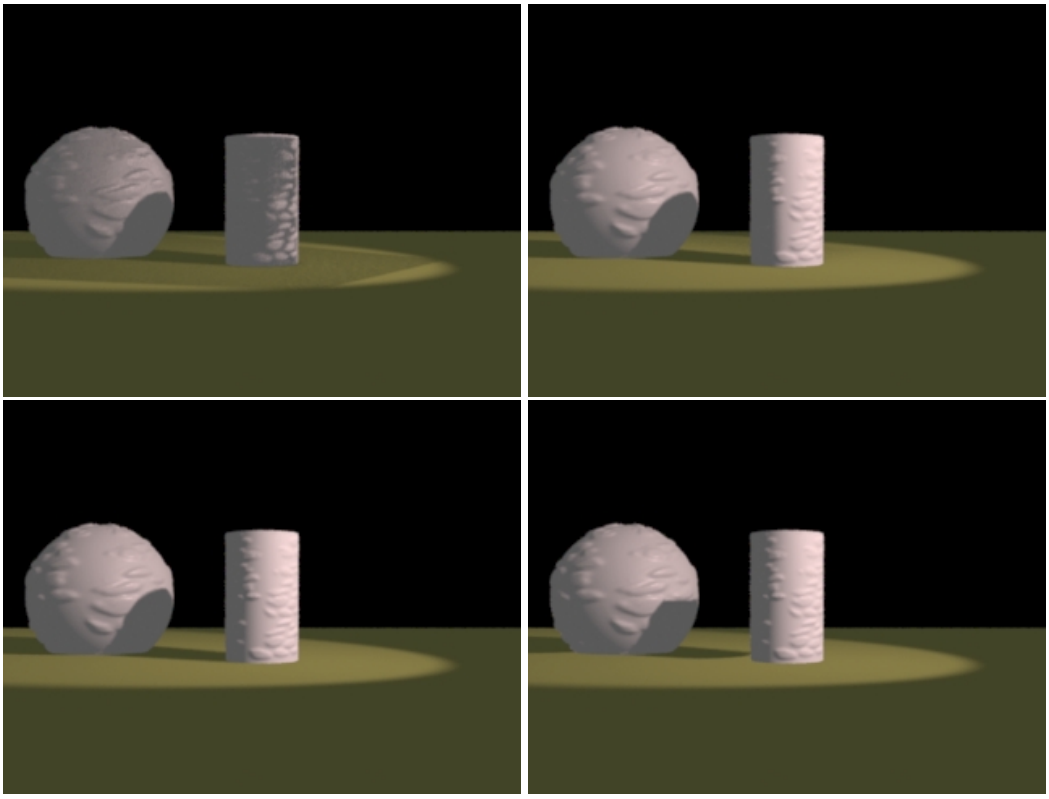


Figure 3.1: Shadow bias: 0.0 (top left); 0.05 (top right); 0.25 just about right (bottom left); 1.0, too much (bottom right).

The zero shadow bias process produces results that are very similar to the newer *PRMan* "midpoint" method. However, it is much more expensive then the "midpoint" method. So generally, one should consider using the built-in "midpoint", over the Zero Shadow Bias technique. But there are some situations for which this technique should be considered, which will be touched upon later.

Here are the steps involved for the Zero Shadow Bias technique:

1. Render two zdepth passes
2. Create a third shadow-map from a weighed average of the two maps
3. Render the color pass using this new processed shadow map.

The benefit, of this method, is that generally one doesn't have to worry about re-rendering due to shadow-bias issues. One common way of attempting to decrease the numerical inaccuracies of shadow maps, is to increase the map size. So another benefit of the zero shadow bias method is that the shadow maps can be smaller.

Here is an example surface shader to be used on the shadow pass that illustrates the technique:

```
/*
 * Example surface shader illustrating Laurent Charbonnel's zero bias shadow
 * technique for SIGGRAPH 2000 RenderMan course.
 *
 * Tal Lancaster 02/05/00
 * talrmr@pacbell.net
 * Walt Disney Feature Animation
 *
 */
surface SIG2k_zerobias (
        float opacity = 1;
        float orientation = 0;
)
{
        uniform color C = color (1, 1, 1);
        normal Nn = normalize (N);
        uniform float val;
        float O;

        val = (orientation == 0)? 1: 0;

        if (Nn . I <= 0) {
                /* point is front facing */
                O = val;
        }
        else {
                /* backfacing */
                O = 1 - val;
        }
        Oi = O;
        Ci = Oi * C;
}
```

The opacity field is used to handle partial transparency or to remove geometry from the shadow computation. Also, one could add more smarts to the shader so portions of the geometry could be selectively removed (e.g., via texture maps or uv coordinates).

The orientation is used to tell the shader to focus on front facing (when orientation = 0) geometry or back facing (when orientation = 1).

Shadow pass one: use the zero-bias shader with orientation set to 0 rendering with the zdepth dspydriver. This makes everything front-facing opaque and everything back facing transparent. The result a zdepth map containing only the front facing geometry. (See Figure 3.2, top left)

Shadow pass two: do the same thing, but with orientation set to 1 rendering out to a different zdepth file. This makes back-facing geometry opaque and front-facing geometry transparent. Saving out a second version of the zdepth information, containing only the back facing geometry. (See Figure 3.2, top right)

The net effect of these two passes should basically be the equivalent of using the newer *PRMan* min/max depth points in the shadows. Shadow pass 1 is getting the min depth values and shadow pass 2 is getting the max depth values. So instead of using the surface shader, one could accomplish pass one by setting the hider option:

```
Hider "hidden" "jitter" [0] "depthfilter" "min"
```

Then for pass2 set it to:

```
Hider "hidden" "jitter" [0] "depthfilter" "max"
```

If this is all that is required then this is all that would be needed and there would be no need for the surface shader. However, if there is a desire that partial geometry be removed for the shadow calculations, then a surface shader containing the necessary code for this object removal would still be needed in the shadow pass.

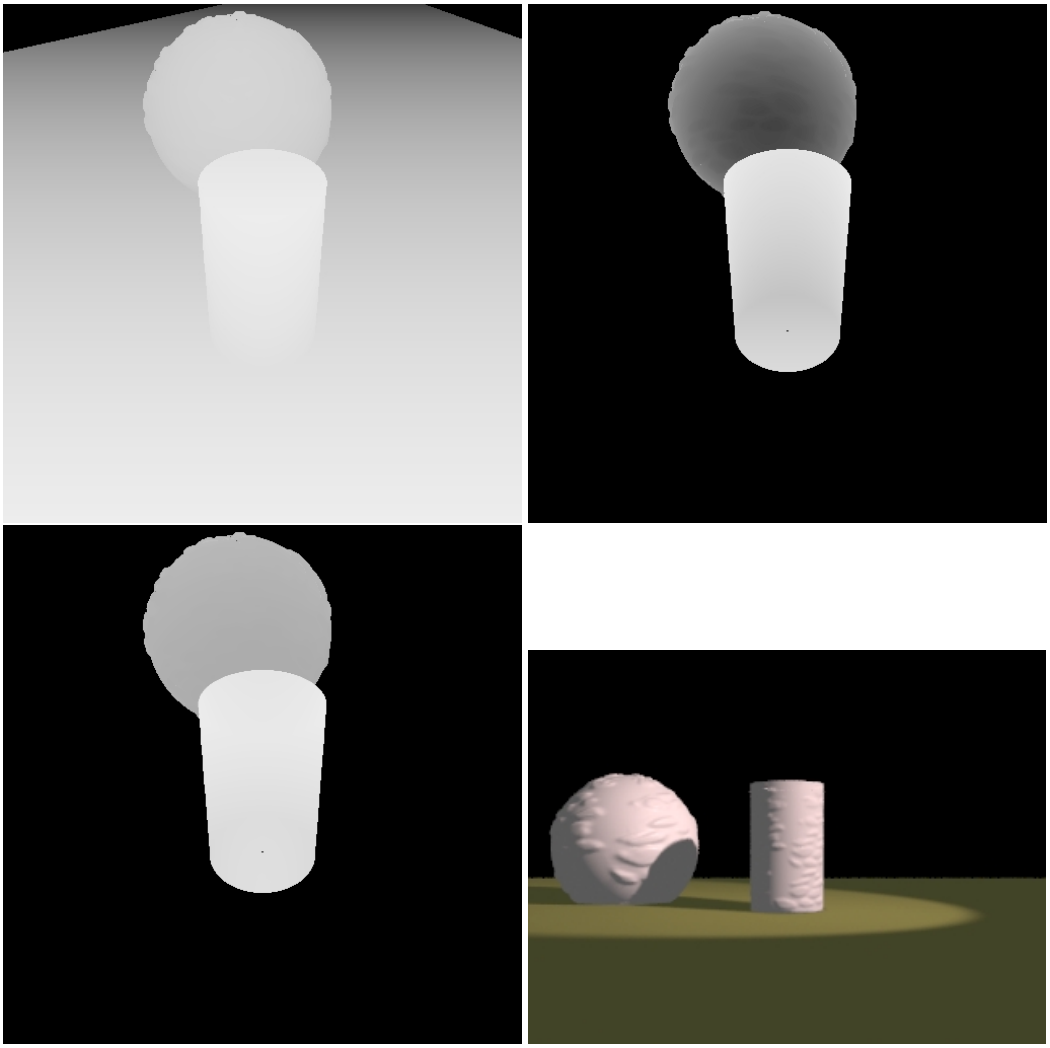


Figure 3.2: Top left: shadow pass 1 (frontfacing); top right: shadow pass 2 (backfacing); bottom left: averaged result of shadow pass 1 and 2; bottom right: Image rendered using zero shadow bias.

Step3: Before the color pass, process the two zdepth files to create a third zdepth file. (See Figure 3.2, bottom left, for an average of the shadow maps from Figures 3.2, top left and right). The exact form of processing could be whatever. On Dinosaur, it was typically a fifty-fifty average between the two maps, which should produce something very similar to the newer *PRMan* mid-

point depth shadow file. But there were controls to perform weighted average of the maps, so one map could have emphasis over the other. This was needed on certain geometry when large blurs were being applied during shadow lookup. The net effect, of the pure fifty-fifty average, was the shadow map was getting pushed back, too far from the front, causing hot spots. By controlling the importance between the two shadow, the problem was alleviated.

Figure 3.2(bottom right) shows the color pass using the shadow map from Figure 3.2(bottom left).

Here is the listing for a C program `SIG_zmean.c`. Its purpose is to apply a straight average between two shadow maps creating a third shadow map.

```
/*
 * SIG2k_zmean.c
 *
 * For SIGGRAPH 2000 RenderMan course.
 *
 * Example program that reads in two PRman zdepth files and creates a
 * new zdepth file that is the average of the two.  This is used to
 * create a shadow map that takes advantage of Laurent Charbonnel's
 * zero shadow bias technique.
 *
 * Author: Tal Lancaster
 * Date: 03/04/2000
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>


typedef struct zfileHeader_t {
        int        magic;           /* type of file */
        short   width;          /* width of map */
        short   height;         /* height of map */
        char    reserved[128];  /* 2 transformation matrices */
} zfileHeader_t;

main (int argc, char **argv) {

    int fin_map1, fin_map2;  /* input maps */
    int fout_map;        /* output map */

    zfileHeader_t zhead1, zhead2;  /* map headers    */
    float zval_map1, zval_map2;     /* depth val from maps   */
    float zval_map3;        /* depth val for new map */
    float *zvals_map1, *zvals_map2; /* depth vals from maps  */
    float *cur_zval_map1, *cur_zval_map2; /* ptrs to current map vals */

    float min_map1, min_map2;   /* min depth val for maps */
    float max_map1, max_map2;   /* max depth val for maps */

    if (argc < 4) {
        fprintf (stderr, "Usage: %s zdepthMap1 zdepthMap2 new_zdepthMap\n",
          argv[0]);
        exit (1);
    }

    fin_map1 = open (argv[1], O_RDONLY);
```

```
    fin_map2 = open (argv[2], O_RDONLY);
    fout_map = open (argv[3], O_WRONLY|O_CREAT|O_TRUNC, 0666);

    /* get depth map header */
    read (fin_map1, &zhead1, sizeof (zfileHeader_t));
    read (fin_map2, &zhead2, sizeof (zfileHeader_t));

    if (zhead1.magic == 0x2f0867ab && zhead2.magic == 0x2f0867ab &&
          zhead1.width == zhead2.width && zhead1.height == zhead2.height) {
        /* both maps are zdepth maps */

        /* store size of maps */
        int mapsize = sizeof (float) * zhead1.width * zhead1.height;
        int i, j;

        /* dump out header information to new map */
        write (fout_map, &zhead1, sizeof (zfileHeader_t));

        /* allocate space for the two input maps. */
        zvals_map1 = (float *) malloc (mapsize);
        zvals_map2 = (float *) malloc (mapsize);

        /* read in rest of map data */
        read (fin_map1, zvals_map1, mapsize);
        read (fin_map2, zvals_map2, mapsize);

        min_map1 = min_map2 = 1e6;
        max_map1 = max_map2 = -1e6;

        cur_zval_map1 = zvals_map1;
        cur_zval_map2 = zvals_map2;

        /* go through maps looking for the max and min values */
        for (j = 0; j < zhead1.height; j++) {
            for (i = 0; i < zhead1.width; i++) {
                if (*cur_zval_map1 < min_map1)
                    min_map1 = *cur_zval_map1;
                if (*cur_zval_map1 > max_map1)
                    max_map1 = *cur_zval_map1;
                if (*cur_zval_map2 < min_map2)
                    min_map2 = *cur_zval_map2;
                if (*cur_zval_map1 > max_map2)
                    max_map2 = *cur_zval_map2;

                cur_zval_map1++;
                cur_zval_map2++;
            }
        }

        /* reset the position, so can go through again */
        cur_zval_map1 = zvals_map1;
        cur_zval_map2 = zvals_map2;

        for (j = 0; j < zhead1.height; j++) {
            for (i = 0; i < zhead1.width; i++) {

                zval_map1 = *cur_zval_map1;
                zval_map2 = *cur_zval_map2;

                zval_map3 = (zval_map1 + zval_map2) / 2.0f;
```

```
                if (zval_map3 > 1e6 && zval_map3 >= 0.5 * max_map1)
                    /* Just a check as the second map could produce
                     * a very large depth value throwing off the mean
                     * so want to clamp it.
                     */
                    zval_map3 = max_map1;

                /* Reusing the map1 data to hold the new map */
                *cur_zval_map1 = zval_map3;
                cur_zval_map1++;
                cur_zval_map2++;
            }
        }

        write (fout_map, zvals_map1, mapsize);

    }
    else
        fprintf (stderr, "Not zfiles or zfiles not same size\n");

    close (fin_map1);
    close (fin_map2);
    close (fout_map);
}
```

The following structure from this program may need some explaining:

```
typedef struct zfileHeader_t {
        int     magic;           /* type of file */
        short   width;          /* width of map */
        short   height;         /* height of map */
        char    reserved[128];  /* 2 transformation matrices */
} zfileHeader_t;
```

This structure reads the header of a PRman zdepth file. The first field, magic, says what kind of file it is. A zdepth file will be 0x2f0867ab. The next two fields: width and height give the dimensions of the depth file. The last field, reserved, gives the two 4x4 transformation matrices (world to light screen space, and world to light camera space).

3.1.1 Pros and cons with this technique

Cons

Using this technique requires two shadow passes, for each light that it is to be used on. It assumes closed surface order. That is front facing geometry is the closest (min depth) to the shadow light view and back facing geometry is the farthest (max depth). Newer versions of *PRMan* shadow depth options (like midpoint) help to alleviate the need for shadow bias tweaking.

Pros

This technique may still be useful in situations where the regular built-in shadow depth options, don't give the desired results. Like in cases where a midpoint isn't what is wanted, but where a weighted average is desired. An example of this maybe if the shadow-maps have very large blurs applied to them. Midpoint will not hold up so well, but zero-shadow bias will by varying the weight of the shadow-maps.

3.2 Camera Projections

Camera Projection[3] is basically a slide projector, where the projector is the camera. One could just project images on a wall (plane). But that is pretty flat and boring. Things get a little more interesting when you apply more complex geometry.

The essence of camera projection is first projecting 3D geometry points to 2D texture points. Then apply a texture to the 3D scene independent of the camera motion. So camera projections provide a quick way to apply a texture-map to a 3D scene. They can be thought of as a form of point referencing (Pref). These projections are also a way to get changes in perspective in 2D paintings.

Various studios have used camera projections for some time now. On Dinosaur, camera projections were used in about 60 shots. The following is a simplified description of how they were used.

What is needed for camera projection?

1. Geometry

   A 3D set is needed. There needs to be something for the textures to project onto. The 3D set needs to contain the major features that are going to be painted. So if there is going to be a hill in the foreground painted, there had better be a model of the hill in the 3D set.

2. Camera

   A frame (or frames) needs to be picked which best represents a particular scene. Frequently, this frame is one that has as much of the scene as visible as possible. Whatever frame(s) is picked, the camera transformations are noted. These frames are known as the camera projection keyframes.

   This technique is based entirely on mapping textures to a particular camera view (the camera projection keyframes). If the camera animation changes so that the camera at the picked frames is no longer in the same position as when the associated keyframes were painted, then this work will be lost. So it is a good idea to not start working with camera projections until the camera animation has been finalized.

3. Renderer

   Ok, you need to use a renderer that is capable of dealing with the camera projection transformations (RenderMan will do). Once the camera projection keyframe is picked, this key (or keys) is rendered out creating keyframe images. It really isn't important what surface shaders are used. Just something that will give a sense of the contours in the scene. So even a plastic or matte surface shader could be used.

   These rendered keyframe images are what will be painted on. Figure 3.3(left) is such an example. It is the CG set, rendered using a matte surface shader, to be painted on.

   The rendering of the keyframes is the first pass. Later, a second rendering pass, of all of the scene's frames, will be rendered using the painted texture-maps with applied camera projections.

4. Paint program (and maybe a painter)

[3] Based on a WDFA internal presentation by Mark Hammel

Figure 3.3: Left: Firey shore CG set. Used for complete digital background. Right: Final CG background, painted on groundplane model. (©Disney Enterprises, Inc.)

Camera projections don't require any special in house software. Nor does it require a full-blown 3D-paint program. A regular 2D program like Photoshop will work just fine. Now the painter paints on top of the rendered camera projection keyframe images. One must be careful so that the major features in the painting match up with the major features on the geometry. This is why painting should happen on top of the rendered frame as it contains the 2D projected information of the scene. If this isn't done then the paint won't project to the geometry correctly, which will be very apparent once the camera and perspective start to change. Figure 3.3(right) is an example of a painted camera projection keyframe image.

5. Rendering CG set (2nd pass)

   Now with the keyframes painted, the shot can be rendered using these maps with the camera projections applied to the CG set.

SIG2k_cameraProjection.sl is a simple, but working camera projection shader. Actually the brains of this shader is found in the function f_cameraProjection() listed later.

```
/*
 * Example surface shader illustrating camera-projections
 * technique for SIGGRAPH 2000 RenderMan course.
 *
 * Simplistic version.  Inspired from shaders by Robert Beech and Mark Hammel.
 *
 * The function that this version calls f_cameraProjection only will work
 * with 1 camera-projection keyframe.  It lacks the sophistication of blending
 * between the camera-projection keyframe maps.  Nor does it make use of
 * zdepth maps for looking for object occulsion.
 *
 *  Some parameters that are used
 *  frame -- the current frame being rendered.
 *  cp_mapBaseName -- The prefix of the camera-projection maps
 *      Utimately the shader expects that the name of the maps follow
 *      a form like testcp.53.tx.  For this example the base name is "testcp.".
 *  cp_mapSuffix -- The tail of the the maps name.  In the example given for
 *      cp_mapBaseName, the suffix would be ".tx".
 *  usePref -- This value should be true, if Pref is provided for the geometry.
 *      This tells the shader to ignore P and use the alternate points instead.
```

```
 *  The Pref points themselves.
 *  Cerror -- This is the color to return if the points being rendered are
 *      outside any of the painted maps.
 *  debug -- Used to invoke the debugging options in the f_cameraProjection
 *      function.
 *  printMatrix -- Set this to true to dump the camera matrix of the
 *      current frame to stdout.  This should be done for each
 *      camera-projection keyframe.  Then have these matrix values stored
 *      in arrays in the matrixdata.h.
 *
 * Tal Lancaster 02/19/00
 * talrmr@pacbell.net
 *
 */

#include "f_SIG2k_cameraProjection.h"


surface SIG2k_cameraProjection (
        float Kd = 0.5, Ka = 0.05, Ks = 0, roughness = 0.15;
        color specularcolor = 1;
        float frame = 0;
        string cp_mapBaseName = "";
        string cp_mapSuffix = "";
        float usePref = 0;
        varying point Pref = 0;
        color Cerror = color (1, 0, 0);
        float debug = 0;
        float printMatrix = 0;
)
{
    point p;
    uniform string cp_CmapFormatStr =  concat (cp_mapBaseName, "%d",
                        cp_mapSuffix);
    color C;
    vector V = normalize (-I);
    normal Nf = normalize (faceforward (normalize (N), I));

    if (printMatrix == 1)
        /* dump out the object->NDC matrix for this camera */
        f_printMatrix ();
    else {
        /* must already have the matrix information, so do projections */

/* This file must define global ARRAYSIZE and arrays keyframes and
    Mobject2NDC */
#include "matrixdata.h"

        /* Get original P before displacement, if it exists */
        if (usePref == 1)
            p = Pref;
        else if (displacement ("Porig", p) != 1)
            p = P;

        C = f_cameraProjection (p, Mcp_obj2NDC, keyframes, ARRAYSIZE,
                    frame, cp_CmapFormatStr, Cerror, debug);

        Oi = Os;
        Ci = Oi * (C * (Ka * ambient() + Kd * diffuse (Nf)) +
                  specularcolor * Ks * specular (Nf, V, roughness) );
        Ci = clamp (Ci, color 0, color 1);
    } /* else not debug */
```

```
}
```

Here are some of the high-points of a camera projection shader:

For the camera projections to work, the camera transformations for each of the keyframes need to be known. This is by done setting the shader's parameter, "printMatrix" to true and then rendering the keyframe using the shader. The matrix information will be dumped out to stdout, via a printf() statement. As the printf() statement will be the same for the entire render, there is no need to render the entire frame. Just kill the render after a few of the lines start printing out.

Here is the code for the f_printMatrix function:

```
void f_printMatrix ()
{
    /* current->object transformation matrix
     * transform ("object", P) same as transform (Mobj, P)
      */
    uniform matrix Mobj = matrix "object" 1;

    /* current->NDC transformation matrix
     * transform ("NDC", P) same as transform (MNDC, P)
     */
    uniform matrix MNDC = matrix "NDC" 1;

    /* object->NDC transformation matrix
     * Pobj = transform ("object", P);
     * Pndc = transform ("object", "NDC", Pobj);  same as
     * Pndc = transform (Mobject2NDC, Pobj);
     */
    uniform matrix Mobject2NDC = (1 / Mobj) * MNDC;

    printf ("Camera matrix = (%m);\n", Mobject2NDC);
}
```

The goal of this f_printMatrix() is to dump out the object to NDC space matrix for a given keyframe. The reason for this is that a camera projection is simply an NDC transformation. Figure 3.4 shows a Point P being transformed into NDC space, Pndc. The variable Mobject2NDC, in the f_printMatrix(), contains this transformation. This is what is dumped out by the function and is used later to perform the camera projection. Note the grey area, in Figure 3.4 is what this keyframe maps to. If the camera moves in any way such that it sees outside this view, then this new view will contain unmapped areas. In such a situation, multiple keyframes will be needed.

The matrix that is dumped out is then stored in the matrixdata.h include file. Note that the shader will need to be recompiled whenever this data file changes. Otherwise the shader won't know about any of the changes.

Now is probably a good time to note that this function will only work with versions of *PRMan* 3.8 and later due to the changes in matrix casts. If you wanted to have this work under *PRMan* 3.7, something like the following would be needed:

```
uniform matrix Mobject2NDC = Mobj * (1 / MNDC);
```

Here is an example of what a matrixdata.h file would look like:

```
/* Define the number of keyframes to use.  Unless, f\_cameraProjections has
 * been properly modified to handle multiple keyframes, this number should
 * always be one.
 */
```

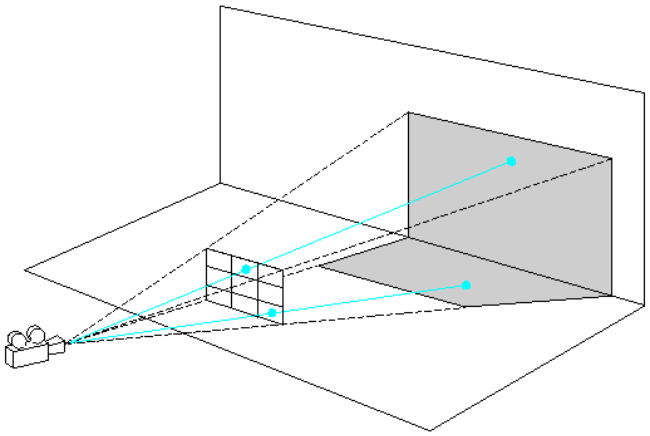


Figure 3.4: Slideprojector NDC projection. Showing a point P, in the scene, being transformed to NDC space, Pndc. The transformation matrix that does this is noted in Mobject2NDC.

```
#define ARRAYSIZE 1

/* store the keyframes that were used.  In this case frame one was the
 * keyframe
 */
uniform float keyframes[ARRAYSIZE] =
  {1};

/* The matrix array of the various keyframes.  In this example the matrix was
 * for frame 1.
 */
uniform matrix Mcp_obj2NDC[ARRAYSIZE];

/* frame 1 */
Mcp_obj2NDC[0] = (1.1798, -0.108326, 0.545303, 0.545303, -0.15267, -2.26076, -
0.294529, -0.294529, 0.239958, 0.136075, -0.784792, -0.784791, 0.54674, 0.62762, 0.987516, 0.9
```

The main workhorse for the camera projection shader is f_cameraProjection(). Its goal is to perform the projection and then return a color. It does this by transforming the point to object space. Then applying the Mcp_obj2NDC transformation matrix of the stored keyframe to convert this point to the keyframe's NDC space. Then the map is read using those NDC coordinates.

Here are some of the high-points for f_cameraProjection().

```
color f_cameraProjection (
        varying point P;
        uniform matrix Mcp_obj2NDC[];
        uniform float keyframes[];
        uniform float arraysize;
        uniform float frame;
        uniform string cpTexFormat;
        uniform color Cerror;
        uniform float debug;
```

```
)
{
. . .
    point Pobj = transform ("current", "object", P);

. . .
    /* Transform from "object" space to the camera projection "NDC" space. */
    Ptmp = transform (Mcp_obj2NDC[i], Pobj);

    C = texture (cp_Cmap, xcomp (Ptmp), ycomp (Ptemp));

    return C;
}
```

Things to watch out for:

Keep in mind that the CP is only good for the field of view of the keyframe. A single camera doesn't always capture all of the visible areas of the scene. So as a camera pans and trucks, there will be null areas (areas outside the painted texture). These null areas are basically points in the scene that weren't visible from the CP keyframe. So there isn't any Pref information available for them. In these situations, multiple keyframes (multiple projections) are needed.

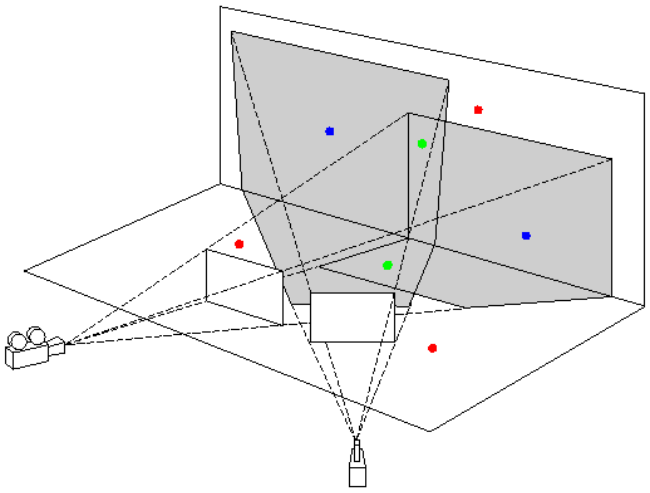


Figure 3.5: Multiple projections. Blue -unique mappings; Green overlapping mappings; Red unmapped areas.

Figure 3.5 shows multiple projections are needed because the camera move has revealed new areas that are in need of mapping. The red dots represent areas that still haven't been mapped yet. As long as the camera never sees those areas, there is no need for them to be painted. The blue areas represent areas that are unique to each map and so present no worries when using multiple keyframe projections. However, the green areas represent areas that are shared between the maps. It is these areas that care is needed to blend (or impose a precedence in the keyframes) between the maps to provide a smooth transition.

A similar problem occurs when the scene's geometry hides other portions of the ground plane. See Figure 3.6. Say the top left image was used as a keyframe. The points hidden by the front dune and the dune itself, share the same NDC projections. So if the camera moved to something like in the middle left image, a camera projection will start to produce a duplicate of the front dune as seen in the middle right image. A solution for this case is to render depth maps, when making the keyframes. Then use the depth maps to aid in determining how to project the maps later when rendering with the painted keyframe maps. This is what was done for the bottom image.

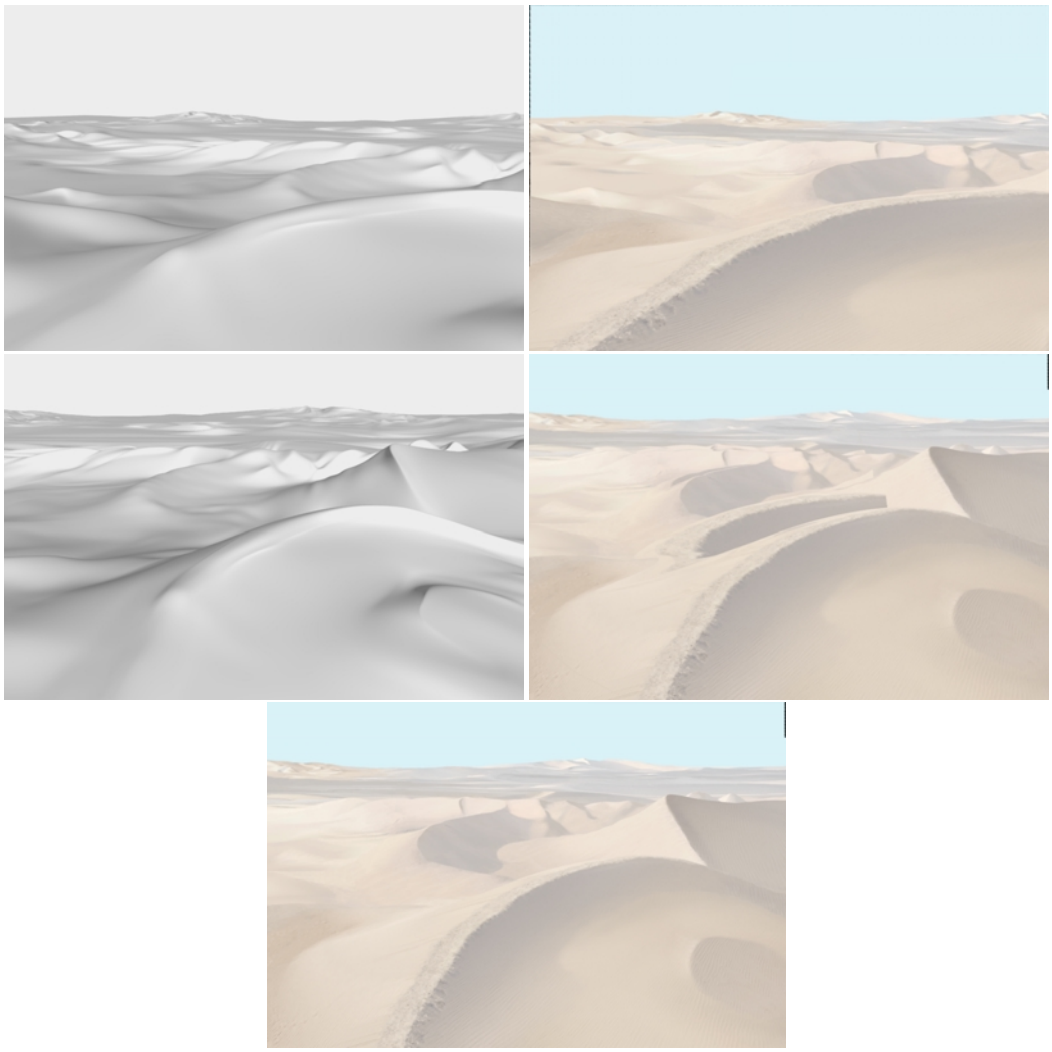


Figure 3.6: Top left: Frame 1 of crescent moon dune CG set. Top right: Frame 1 of crescent moon dune. Middle left: Frame 81 of crescent moon dune CG set. Middle right: Frame 81 without zdepth map option. Bottom: Corrected CP using depth maps. All images ©Disney Enterprises, Inc.

An example of another situation to watch out for, say the scene has a very long truck out and the last frame of the scene is used as the CP keyframe. The very early frames, in this situation, will have a very low-res quality. This is because the texture map will be using a very small sub-portion of the CP keyframe texture-map. A solution for this situation is to have a second CP keyframe near the beginning and then blend between these two maps (or impose a map precedence).

Figure 3.7 shows an example of a severe pullback. The top left image (a) shows the first frame of the camera animation. The top right image (b) shows the final frame of the animation. Figure 3.7(a) represents just a few percent of (b). So if just one keyframe was picked (b) 19), the earlier frames in the animation will be very low quality, because they represent such a small area of the image. A solution is shown in the bottom image, where a frame earlier in the animation is picked as another keyframe (the white area numbered 1). This way the earlier frames can use a map with more detail for their level. Then the later frames can make use of the second keyframe. Care will be needed when transitioning between the maps. This may include blending along the borders of the maps so they will merge seamlessly.

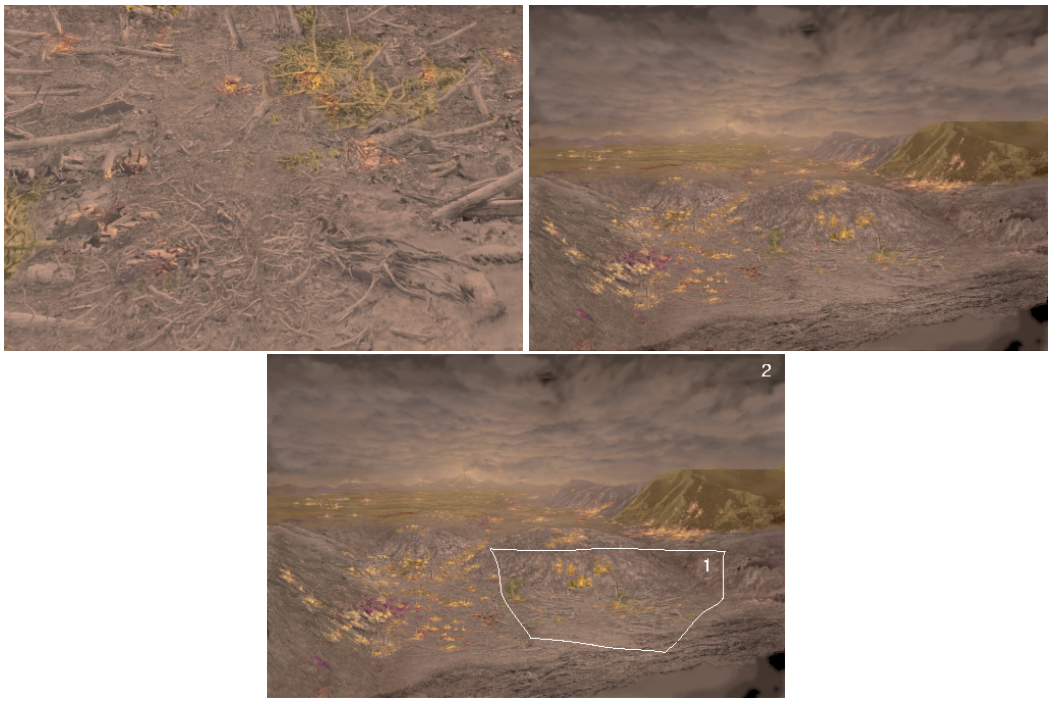


Figure 3.7: Top left: Beginning frame of camera pullback. Top right: Final frame (367) of camera pull back. Bottom: This image, numbered two, is the area seen by the second keyframe. The white area, numbered one, represents the area seen by the first keyframe. ©Disney Enterprises, Inc.

As mentioned earlier, it doesn't matter what is painted as long as the CG set has the key features that represent what is to be painted. There are other uses for this technique, than just painting full digital sets. Another way that this technique was used, on Dinosaur, was to help clean up some of the live action plates. Frequently, these plates had to be cleaned to remove various unwanted objects.

You know to remove things that didn't exist 60 million years ago, like tracking balls and roads. Traditionally, these frames are cleaned up by hand one frame at a time. Under the right conditions, camera projections can make this task much easier.

Say for example, that the camera move is just a slow truck out. Create an exact model set from the shot footage (if you don't already have one). Use the plates from the shot footage to paint keyframe images. Then the only plates that need to be cleaned are the ones used to paint those keyframe images. Figure 3.8(left) shows the original plate around frame 154 of a 280 frame shot. Figure 3.8(right) is has been cleaned by rendering the CG set using frame 280 as the CP keyframe (obviouslly Figure 3.8(right) hasn't been comped with the final background, but you get the idea). In other words, only frame 280 was cleaned up by hand. Camera projection took care of the rest.

Figure 3.8: Left: Original plate around middle of camera move. Right: Cleaned up frame via camera projection. ©Disney Enterprises, Inc.

Camera projections don't have to be used exclusively for painting color maps. This process could also be used to create maps for displacement shaders. Also, transparency could be used to help muddle up the geometry too.

The pluses of using camera projections: CP is less work then 3D paint because only one (or small number) paint job is needed. Also, it will match up exactly to Layout[4] because it is using the actual camera from the scene. It doesn't have the same hang ups as 3D painting, such as perspective distortions of geometry. With camera projection, the camera view is perspective while painting under 3D is usually orthographic. In orthographic it can be hard to determine how features should distort in different views. This is especially true when the sets are very large and the perspective distortion is huge. Don't forget camera projections usually require less work than traditionally cleaning up live action plates.

In conclusion, CP is a very simple technique that makes certain tasks much easier. But it is very limited in its application. It doesn't do well when the perspective shift is extreme. There are just times where painting the scene with 3D painting will be needed.

[4]Layout is the department in the production pipeline that puts together the models, sets, and cameras. The purpose is for staging how the character relate to each other and where everything will be in frame.

3.3 Using RenderMan as a modeler

How would you build a detailed model of a 20-foot Dinosaur? The model will need to have different kinds of scales for skin; pads on the bottom of its feet; worn toenails. Oh, did I mention that it would need to be able to hold up to being scrutinized at less then a foot away and at resolutions up to 2K? In other words this model has to be hyper-real.

Once you are done with that model, there are another 41 more to go. On Dinosaur we had 42 different characters, totaling 18 different species.

The need for this super high detail was that there was a strong desire to have character interactions. If you have a three-foot high lemur walking or sitting on a sixty-foot high dinosaur, and the camera is fairly close up on the smaller character, the detail on the larger character really needs to hold up.

Exactly how high a detail are we talking about? Check out Figures 3.9–3.12.

Figure 3.9: Basic model of Carnator. ©Disney Enterprises, Inc.

Figure 3.9 shows the basic model of the carnator. This is what the model looked like coming out from the modeling department. Now check out Figure 3.10. This figure has the displacement maps and displacement shader applied to the model. Do you notice any differences? Check out all those horns! There are three extra rows of horns that are not in the model and don't forget about all those scales. Figure 3.11 is with the color layer added. Ok, now check out Figure 3.12. This is how close we had to get and they still hold up! Look at the detail on each of those scales.

Attempting to reach this level of detail with traditional modeling techniques would have been very daunting. Our models got heavy very quickly. Four hundred NURBS patches on a model was not uncommon. Some of them even had more than double that amount. Even so the models still were far from the required detail.

Figure 3.13 shows the basic Aladar model. The lines on the model attempt to give a sense of the

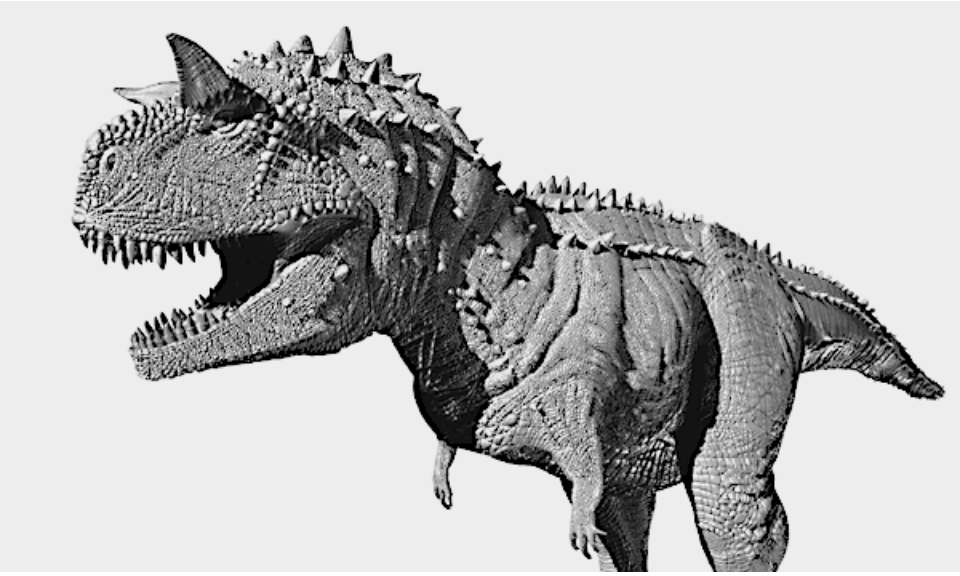


Figure 3.10: Greyscale Carnator with displacements. ©Disney Enterprises, Inc.

patch boundaries, to give a feel of the size and number of patches on the model. Even with those 380 patches, the look is still lacking, considering the camera may (and does) get to within inches from his face.

For us, the solution was to have our models contain the grosser features and then use *PRMan* displacement shaders to add the finer features that would be harder to do in modeling.

It made sense to do it this way. We had to show several versions of these finer grain details for each model. Through the use of displacement maps, these finer grain details can be changed logarithmically faster than they could be done in a modeling program. A texture painter just doesn't have the same issues that a modeler has. The texture painter just paints the pattern that they want. So any change, in the features, that is wanted happens as quickly as the painter can paint it (well ok, paint and then render). Of course, the painter does not have as much control as the modeler, which is why, we still used the modeling process to achieve a good approximation of the final character.

If one attempts to put all of the details, in the model, directly other considerations need to be given when the model starts to bloat. These are mainly connected to the increased I/O in dealing with the bloated model. A very heavy model will slow down interactively for the animators; increase processing time for muscle and skinning; increase load on the paint program (i.e., may only be able to load in less and less of the model at a time); longer render times, etc.

At one point we weren't so sure about using the displacements, so there were some tests done to put more detail in the model. One of them involved adding detail to one of the model's head. In the end, the head had 10X the number patches and it still didn't match up to the detail that we were getting with displacements.

Even if the time involved in building the model didn't matter and ignoring the other issues associated with bloated models, it still may have been impossible to have built such a hyper-real detail model at all. For fun somebody tried to convert a model to polygons based on the data from

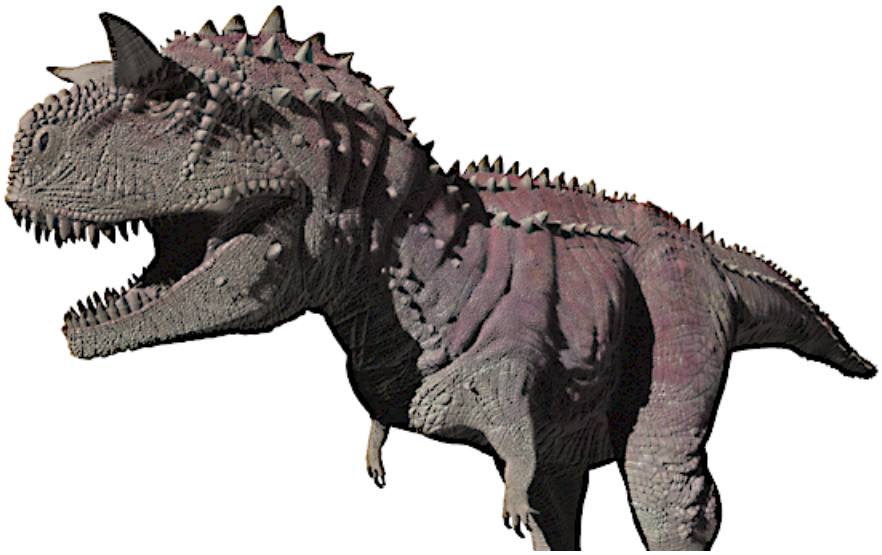


Figure 3.11: Carnator with displacements and color. ©Disney Enterprises, Inc.

some of the displacement maps. They ended up so many polygons that the modeling program and machine (which had 2G of RAM) choked up. Even at this point the model was far from being fully processed.

Let's say you did manage to get all of this super high detail into the model and you manage to get it through your pipeline and can actually render it. What do you do when the model starts to alias? This is geometric aliasing that we are talking about. Under *PRMan* you have no recourse but to drop the shading rate and bump up the pixel samples. Surely, bumping up some already very high rendering times.

Whereas leaving this detail to displacement, one can apply anti-aliasing shading tricks to remove the required detail. Using displacement maps can be thought of as a form of lazy TD's level-of-detail. The multi-res maps of *PRMan* pick just the right size detailed map for you automatically and quickly. So all of the higher frequencies in the maps will be removed first. As long as the model is going to be at least a couple of hundred pixels in size, displacement mapping can provide a very convenient level-of-detail automatically.

Now is a good time to bring up an important point. Everybody knows that *PRMan*'s texture look up function is great at anti-aliasing. Whether, the map is being used for color, bump, displacement, etc., the texture function does indeed anti-alias the values it reads in, very well. There is a problem though; when these texture values are used for displacements. Something more is needed when dealing with high frequency texture maps to displace geometry. "Why?" you may ask, "You just said that the texture function anti-aliases my texture just fine." The crux, of the problem, is that small changes in displacement maps can effect the surface's reaction to light non-linearly! The regular filter from the texture call, albeit-it a nicely anti-aliased one, doesn't address this issue. For this the reason, one is apt to see buzzing when high frequency maps are used for displacement.

Laurent Charbonnel, a Lookdev TD on Dinosaur, came up with a great function that we used

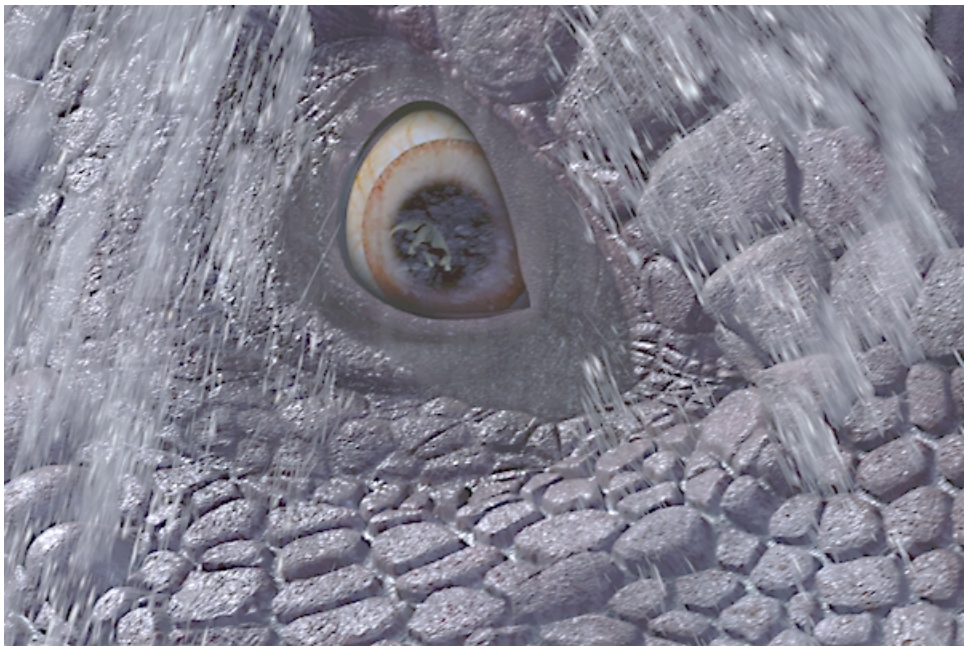


Figure 3.12: Carnator eyeshot (in rain). ©Disney Enterprises, Inc.

in looking up our texture maps in *PRMan*. In the way that it filters, this function tries to lessen the impact that minute displacement shifts will have on the surface's reaction to light. With Laurent's function we can control the amount of buzzing that is acceptable on a per patch basis (and a per shot basis as lighting changes affect displacement aliasing). The result is this function removed A LOT of buzzing in our textures. Thus making displacements viable for our production.

Keep in mind that on Dinosaur we used *PRMan* 3.7. So we didn't have access to newer features in *PRMan* like the radial-bspline texture filter. This newer filter does do a pretty good job of filtering the textures for displacements. It is a good general solution with very little user intervention. So if you don't have anything else to use, then you really should be using this texture filter when reading texture maps for displacements.

Instead of displacement-mapping, there is always bump-mapping. But there may still have been some of the buzzing, due to the same problem. Also, we would have lost all of our nice profile detail. The bump-mapping would have made the profiles smooth and details would shrink and grow depending on the orientation of the camera. Don't forget those horns on the carnator. With bump-mapping we would have lost them, too. No, for us displacements were the way to go.

So it made sense to add the rest of the detail, like the Dinosaur's scales and folds of skin, with *PRMan* displacements. By going with the displacements, we were able to tweak the height of the scales (or even particular scales) on a shot by shot basis. Sometimes the directors just wanted the detail lessened or accented in certain shots. Such changes were accomplished very easily through the use of displacement by either changing the displacement scalar value for the shader or using a mask texture layer to alter the amount of displacements for that area.

On Dinosaur, all of the texturing was done in layers. There were several reasons for this.

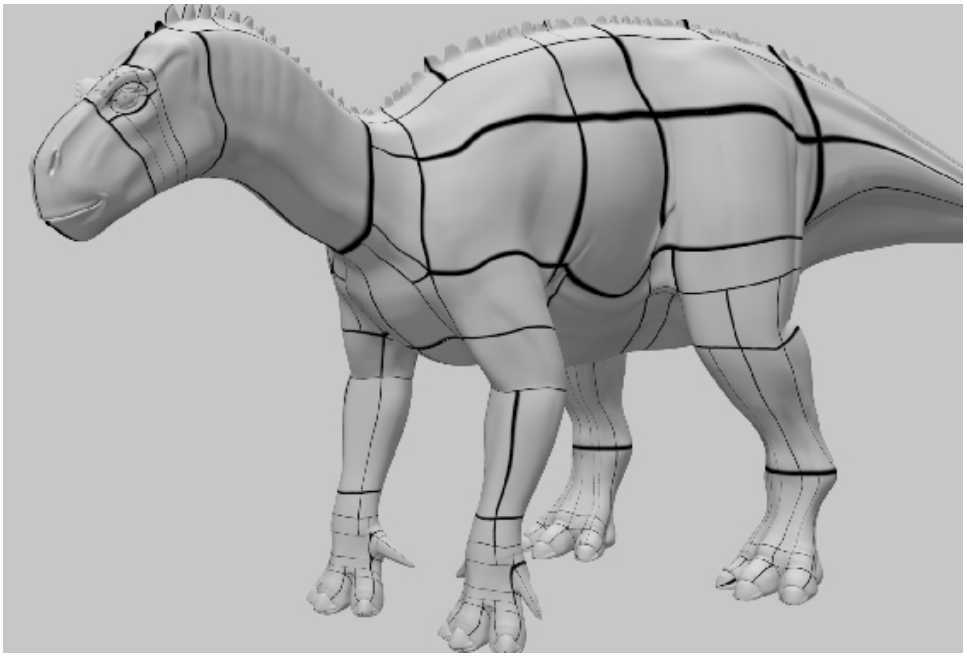


Figure 3.13: Basic Aladar model. The lines are trying to give a sense of the model's patch boundaries. ©Disney Enterprises, Inc.

One of these reasons was so changes to one layer would not impact on another. Generally, there were at least two displacement layers on a character. One layer created larger features like folds of skin. Another displacement layer created the pebbling and scales on the dinosaur's skin.

Another reason was so each layer could be controlled individually. That way, we could play with the balance between different types of displacements by number tweaking in the shader. Doing such tweaking in the shader is much faster than trying to repaint, as the painter has to try to estimate relative grey levels.

So that one displacement map could both displace in and out, middle grey (0.5) was picked as the center of the map. Values less then 0.5 would displace inwards. While values greater than 0.5 in the maps would displace the surface outwards. A big benefit is the displacement bound is half in size. What this means is that you get the same amount of detail for half the displacement bound. By having the displacement bound smaller, the rendering times should be smaller and there should be less opportunity for displacement cracks. Also, this way of painting is conceptually easier for the painters as they could either cut in or out rather then just always expanding the model outwards. The downside is that now the painter only has 128 levels to paint in a given direction rather than the usual 256 (There are still 256 levels from bottom to top of course, but only 128 to cut in, 128 to cut out).

There were lots of attribute maps to control various effects, like dust, wetness, calluses, scratches, and wounds. These characters were suppose to look like they lived in their environment. So they had dirt in between their scales. They were covered with dust. They had worn toenails and calluses on their knees and elbows and other worn areas. They had scratches and scars from old injuries.

For immediate injuries that happened during the film there was the wound layer. The wounds were another displacement layer. The purpose, of this layer, was to give the effect of torn and bitten flesh. It could remove features from the other displacement maps.

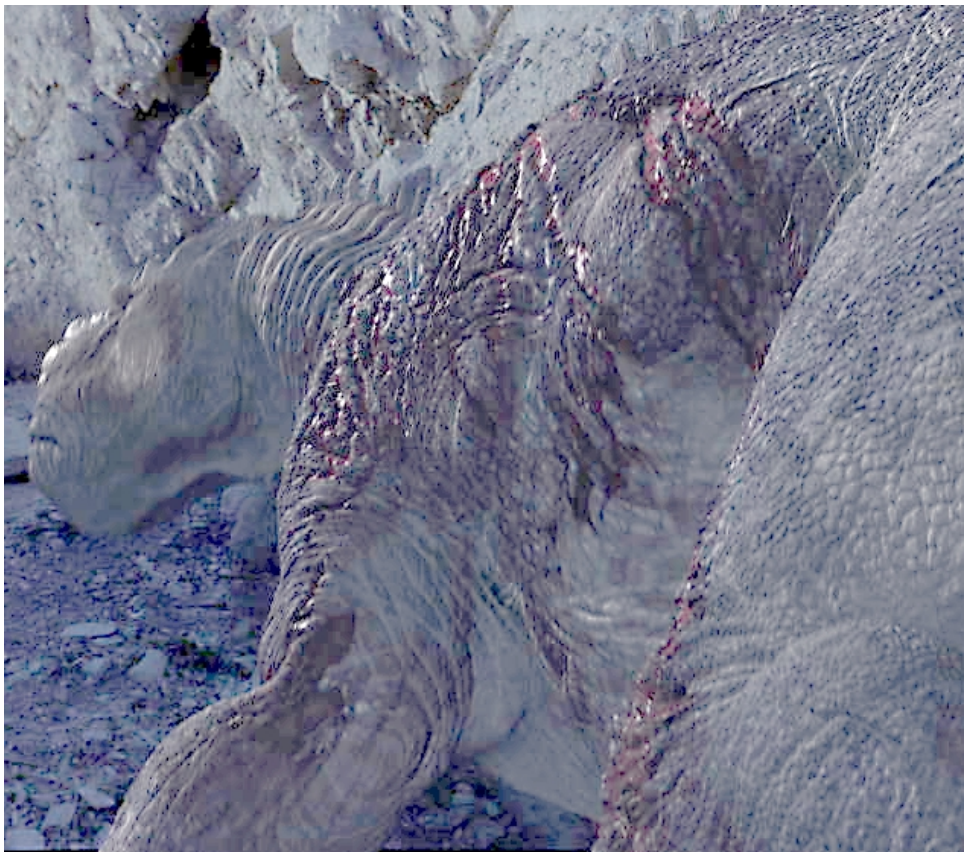


Figure 3.14: Wounded Bruton after Carnator attack. ©Disney Enterprises, Inc.

Downside

However there is a price to be paid for abusing displacements like we did on Dinosaur. One penalty was with disk space. The textures consumed Gigabytes of disk space. The paint program that we used created a texture map for every patch. As we painted using lots of different layers, the number of texture maps was tremendous. Take our example of a 400 patch model $\times$ 5 layers which brings us to 2000 maps for just one model! Some models had up to 11 layers and some models had a lot more than 400 patches. Also some of the maps for these layers can be quite large. Fortunately, disk space is cheap.

Another price with using such severe displacements is that rendering times will start to pick up when the camera starts getting real close to these displacements. So it is a good idea to try to

Figure 3.15: Plio with wounded Bruton. ©Disney Enterprises, Inc.

maximize the efficiency of *PRMan*, by getting the displacement bounds as tight as possible.

It has been my observation that it is common enough to find displacement maps that don't go all the way to pure black or pure white. So frequently it is worth going through the maps to find out what their maximum ranges really are. Then use this in factoring the displacement bounds for each of the patches.

Wrap-up

We did what we did because of the hyper-real detail requirements that were needed for this movie. Take the lemurs for instance, they had pads on their hands and feet. These pads had to hold up to being viewed at 2k. Plus they interacted with characters twenty times their size. The detail on these much larger characters had to hold up when these smaller creatures touched, sat, and walked on top of these larger ones. All of these characters abused *PRMan* displacements, like they have never been abused before. Remember the carnator!

So there you have it, a small taste of the rendering related issues on Dinosaur.

Chapter 4
The Artistry of Shading: Human Skin

Mitch Prater
Pixar Animation Studios
mitch@pixar.com

4.1 Approach

4.1.1 Artistic v.s. Scientific

Past attempts at simulating human skin were often based on direct simulation of the physical properties of skin. This approach begins with study and data collection of the illumination properties of skin and ends with models describing the interaction of light with and within the various physical components that make up skin, and how all those combine to yield the final result. Depending on the level of accuracy of the data collection, the models derived from the data, and the simulation environment, this can result in a simulation of skin that has some very realistic looking qualities.

This is all very good, solid, fundamental science. And it helps us a great deal to understand the nature of skin. But to an artist, this approach seems a bit like examining a person's DNA in order to figure out how many freckles are on his nose. Except for certain schools of modern art, to an artist it's all about the end result, and the ability to control that result, rather than the process used to arrive there - even if the end result is something that's supposed to look as realistic as possible.

If a painter is very skilled, he can create a representation of skin that is extremely realistic, even though paint on a canvas has vitually nothing to do with the physical processes of light interacting with skin. The reason this is possible is a result of something every artist knows fundamentally: that we will see what they want us to see as long as they are able to capture the fundamental essence of how we percieve something, and recreate it.

So my approach was to study skin in order to develop a sense of what makes skin be percieved as skin, and to reproduce those qualities in a way that gave us control over them, as well as incorporating physical characteristics when doing so would enhance the perception.

4.1.2 Reference Material

Unlike a scientific approach, where I would have read lots of research papers, disected skin to understand how it's put together, and taken lots of reflectivity measurements of its various constituents, I took the artistic approach and just looked at lots of pictures that conveyed various aspects of skin, or of skin as a whole, very well. From this, I determined what qualities we wanted and how they would all fit together into a single conceptual structure.

4.2 Conceptual Structure

4.2.1 Controllable Features

The skin had to be applicable to the full range of race, sex, and age. The properties of the skin itself had to be manipulatable to accomodate this variety as well as allow the addition of optional features such as makeup, whiskers, and sweat. On a practical level, the skin had to also change between different types of skin, allowing a smooth transition into the mouth and ears. All of this variability was grossly controlled with settings to adjust the overall characteristics, as well as with paint to provide detailed local control. The features themselves were chosen to be easily identifiable, both physically and aesthetically, and to be mutually orthogonal. Any interaction of the controls, as is often the case with a prurely physically based implementation, would make the controls completely impractical. These were probably the most crucial aspects of the skin: that all characteristics were defined to have a direct physical and aesthetic meaning, that all characteristics were visually orthogonal to each other, and that each was specified using a normalized range that remained consistent in its effect throughout its range and the ranges of other features.

4.2.2 Layering

Each feature applies to a specific layer of the skin, whether within the skin itself or on its surface. The layering reflects the actual physical layering of the components themselves. Since all the characteristics were carefully constructed using a normalized range, and to be orthogonal to each other, there was no difficulty in specifying and controlling each characteristic individually, even though in reality they may reside in the same physical layer. Other features would occlude features in lower layers depending on their physical properties of opacity and the degree to which they were applied. This layering of features applied to the illumination of each feature as well, since different features often had very different responses to illumination.

4.3 Implementation

4.3.1 Data Collection

One of the keys to realistic skin is the accurate reproduction of the natural, detailed variation found in human skin. This is particularly true when the skin is to be viewed from very close range, as was needed in Toy Story 2. To collect a sample of this variation, the facial skin of a suitable subject as placed on a scanner, and a high-res scan taken. Various types of skin were scanned in order to acquire samples of pores and whiskers in addition to ”clear” skin. The scans were manipulated

in a paint application to create tiling textures covering a suitably large area so as to preclude any noticable repetition. These texture maps formed the foundation for all the basic variation found within the skin itself.

4.3.2 Manipulation

While, in effect, all controls had to be orthogonal, to achieve this result the internal functionality did not necessarily operate orthogonally. In fact, most of the controls for the various features were very carefully coupled together in such a way as to maintain a normalized range of effect. One example of this was ”blotchiness”. In order to normalize the effect of this control and thereby decouple it from the skin color, its effect on the color variation was modulated by the overall color; since, as the color became darker, the range of color variation to achieve the same degree of percieved blotchiness needed to be decreased dramatically. ”Shininess” and many illumination characteristics were similarly manipulated to maintain a consistent and perceptually orthogonal effect relative to other features.

All of the skin qualities were set to default values according to the race, age, and sex of skin that was to be created. These default values were all locally modifiable through the application of paint to the individual human model. In this way, local variation such as the application of makeup, wrinkles, oiliness, whiskers, pore characteristics, sweat, and other skin features were added to each individual. The quality and detail of these paintings, coupled with the level of control afforded by the skin model, were a major factor in the realism of the skin.

4.3.3 Composition

All the layers were calculated and illuminated individually. Once the final result for each characteristic layer was determined, the layers were composited together using standard compositing algebra to arrive at the final skin result. Generally, the opacity used was derived directly from the paint pass used to control a particular layer. In the case of layers under the skin, this opacity value was further modulated by a function that described the variation in the density of the skin.

4.4 Results

The following images show the results of this work, implemented on human characters in the film *Toy Story 2*. The first image is of Al McWhiggin (sans his distinctive facial hair). The second image is of the toy cleaner, otherwise known as Geri from *Geri’s Game*.

Figure 4.1: Al. Image ©2000 Disney/Pixar, All Rights Reserved.

Figure 4.2: The Cleaner. Image ©2000 Disney/Pixar, All Rights Reserved.

Chapter 5
Fur in *Stuart Little*

Rob Bredow
Sony Pictures Imageworks
rob_bredow@yahoo.com

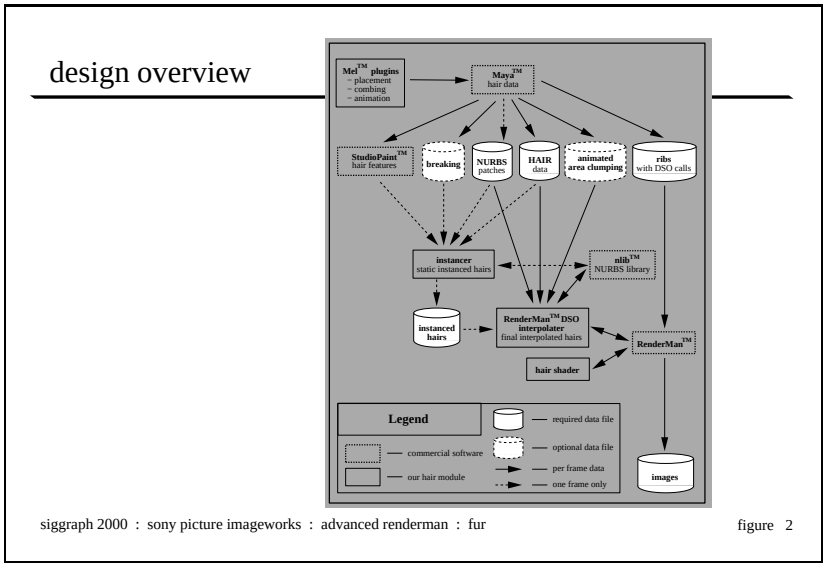
Abstract

Realistic fur instantiation and shading is a challenge that was undertaken during recent work on the film *Stuart Little*. This talk will detail some of the techniques used to create Stuart Little's furry coat and how RenderMan was used to light and shade the mouse's hair.

5.1 Goals

- Must be cute (and realistic).
- Retain detailed looking fur regardless of size on screen
- Realistic lighting model
  - needs flexible controls for cases where realism is not desired
  - handle multiple lighting environments (day, night, sourcey, diffuse)
  - wet/shiny/dirty
- Fur interaction with environment
  - wind
  - other objects
  - self
  - clothing
- Multiple furred characters on screen at one time
- Efficient enough for nearly 500 shots

5.2 Design Overview

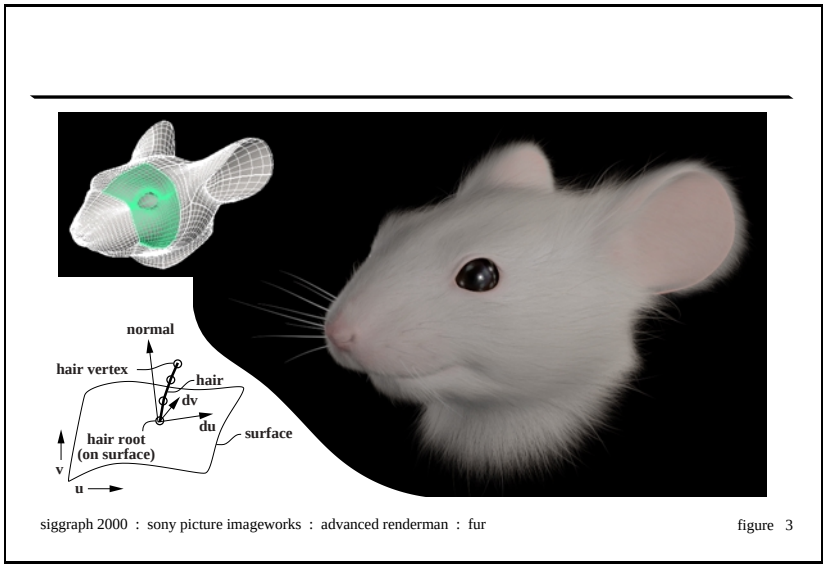


The general pipeline consists of two major processes, those that happen once for each character and those that happen once for each frame. The former consist of gathering all of the data which controls the look and contouring of the fur and calculating as much as possible to be stored in a series of data files. The second process consists of the actual creation of the fur on the character from these data files at the time of the render.

Most of the data for the first set of processes (to design the look of the hair) is managed through MEL scripts in Maya and attributes controlled through StudioPaint. The instancer is then run to compile all of the appropriate data files into a “instanced hair” data file which specifies the location of each hair on the character. This step only has to be calculated one time at the initial setup of the character or if there is a change to the fur data files.

It is important in the design of the system to leave as much of the large dataset creation until the latest possible point in the pipeline. So, the actual RiCurve primitives are not “grown” onto the surfaces until the time of the render when RenderMan notices the DSO references built into the rib file and calls the interpolator DSO. The DSO then generates the rib data which contains an RiCurve for each hair contoured to the surface appropriately and taking into account the character’s animation. The hair is then shaded with a standard RenderMan shader and the output image is generated.

5.3 Fur Instantiation (the interpolator)



The interpolator performs the function of “growing” the RiCurve primitives from all of the input data.

- instanced hair data

  The instancer specifies the position of each of the individual hairs for each patch in terms of the u, v coordinate of the surface.

- animated nurbs patches

  These patches form the skin of the character and are used to position the hair in the correct 3d space and orientation according to the surface normal, and the surface derivative information at the position of interest.

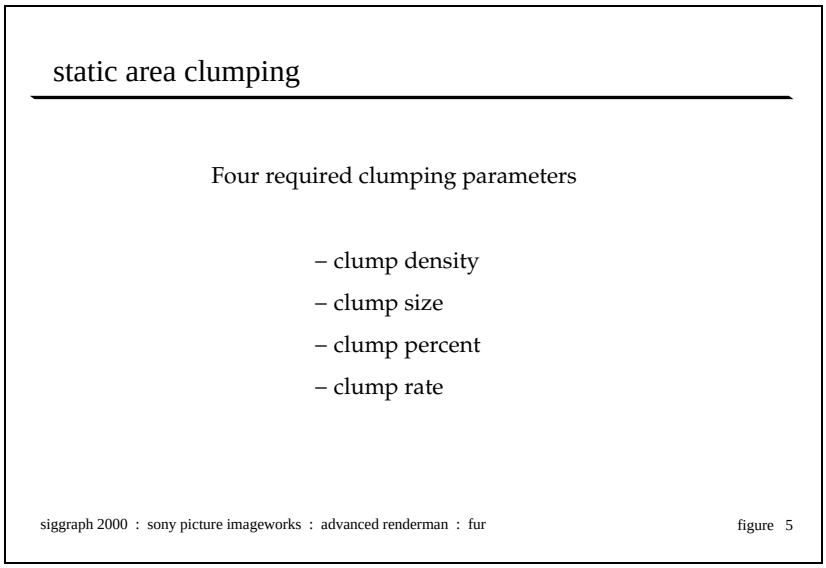
  item hair data from Maya and optionally animated area clumping information.

  This data is used to generate the 4 point curve taking into account the combing information and the other fur attributes specified in the data files.

5.4 Clumping

Clumping of hairs can occur when the fur gets wet due to the surface tension of water. The overall effect is that sets of neighboring hairs tend to join into a single point creating a kind of cone-shaped “super-hair” or circular clump. We have implemented two kinds of clumping techniques: s*tatic area clumping* and *animated area clumping*. The former generates hair clumps in fixed pre-defined areas on the model, whereas the latter allows for the areas that are clumped to change over time. In both cases, we provide parameters which can be animated to achieve various degrees of dry-to-wet fur looks which can also be animated over time.

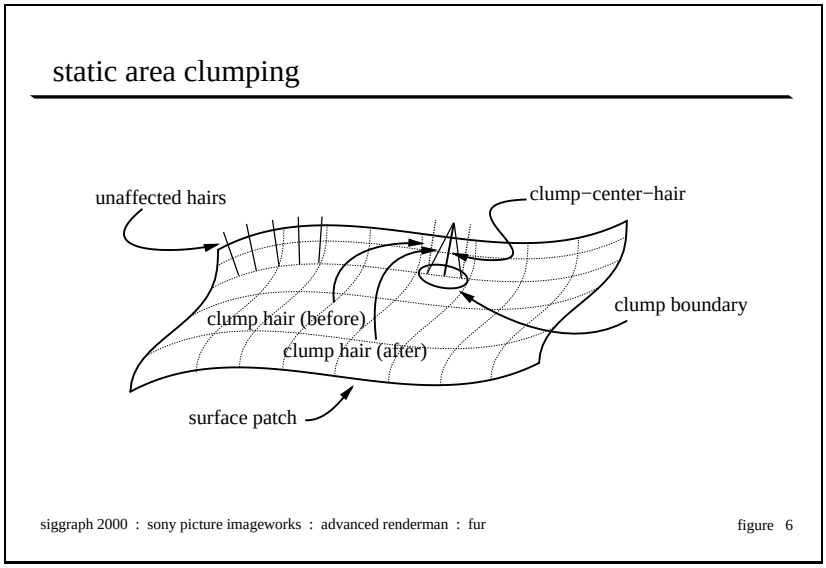
5.4.1 Static Area Clumping

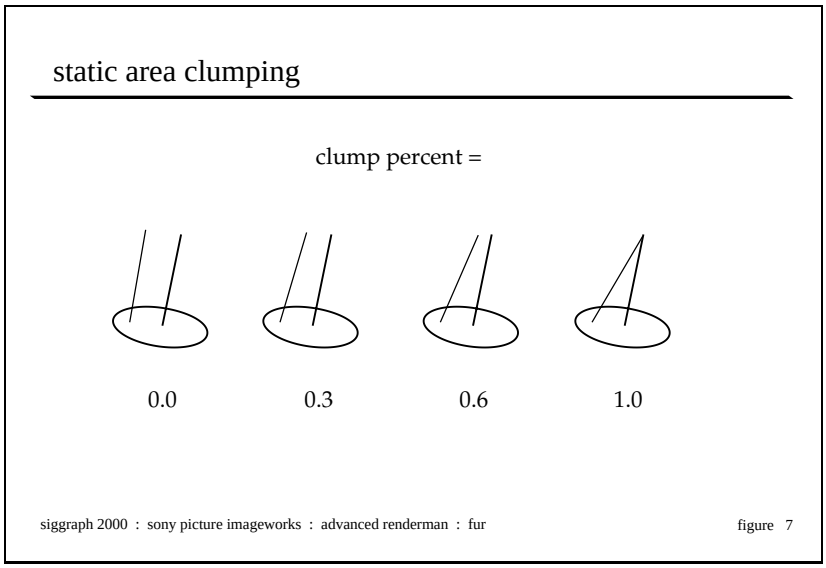


There are four required clumping input parameters: clump-density, clump-size, clump-percent and clump-rate.

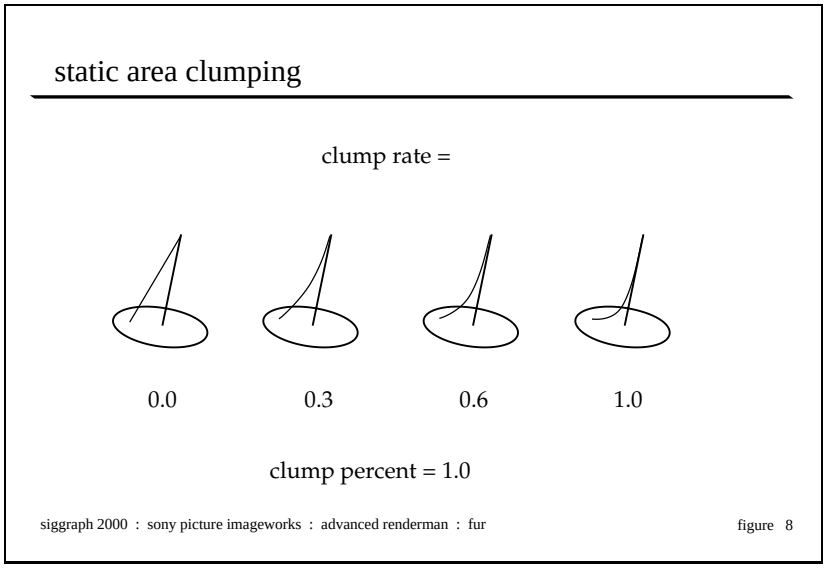
Clump-density specifies how many clumps should be generated per square area. Our clump method translates this density into an actual number of clumps, depending on the size of each surface patch. We call the center hair of each clump the clump-center hair, and all the other member hairs of that clump, which are attracted to this clump-center hair, clump hairs.

Clump-size defines the area of a clump in world space. To determine clump membership of each final hair, our clump method first converts the specified clump-size into a u-radius and v-radius component in parametric surface space at each clump-center hair location. It then checks for each hair, whether it falls within the radii of a clump-center hair; if so, that clump-center hair's index is stored with the hair; if not, the hair is not part of any clump, i.e. is not a clump hair. Our clump method also has an optional clump-size noise parameter to produce random variations in the size of the clumps. Furthermore, texture maps for all four parameters can be created and specified by the user, one per surface patch, to provide local control: for instance, if clump-size maps are specified, the global clump-size input parameter is multiplied for a particular clump (clump center hair) at (u,v) on a surface patch with the corresponding normalized (s,t) value in the clump-size feature map for that surface patch. Also, a clump-area feature map can be provided to limit clumping to specified areas on surface patches rather than the whole model.

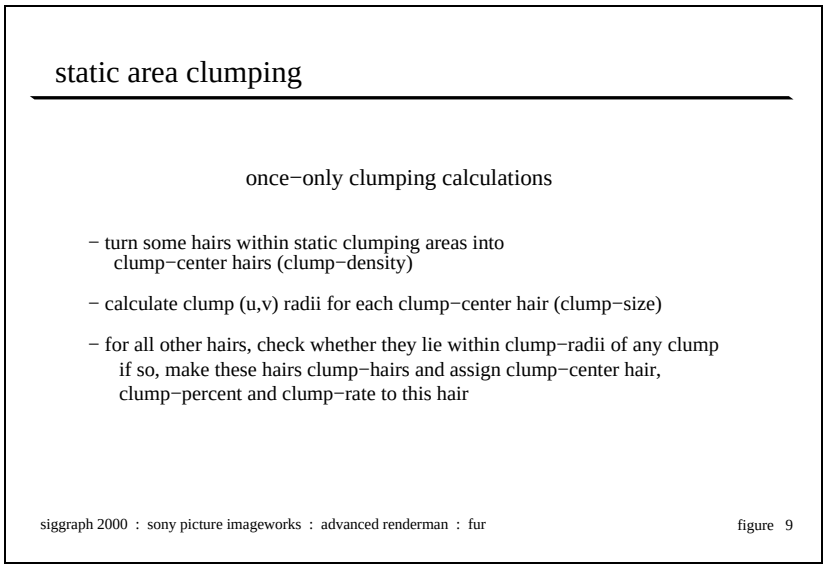




Clump-percent controls the amount the hair is drawn to the clump-center-hair. This is done in a linear fashion where the endpoint of the hair is effected 100% and the base of the hair (where it is connected to the patch) is never moved. Higher clump-percent values yield more wet looking fur.



Clump-rate is the control used to dial in how much the fur is curved near the base to match the clump-center-hair. This is also interpolated over the length of the fur so the base of the fur is never effected by the clumping procedure. Dialing this parameter up yields dramatic increases in the wetness look of the character as the individual hairs are drawn more strongly to the clump centers.



static area clumping

for−each−frame clumping calculations

− generate all clump−hairs
(default for "numberOfCVs" is 3, and "i" ranges from 1−3)

```
fract = i / numberOfCVs;

delta = clumpPercent ( fract + clumpRate * ( 1 − fract ) );

clumpHairCV[i] =     clumpHairCV[i] + delta *
            ( clumpCenterHairCV[i] − clumpHairCV[i] );
```

siggraph 2000 : sony picture imageworks : advanced renderman : fur figure 1 0

The equations for calculating the clumped vertices are illustrated above. The vertices of each clump hair (except the root vertex) are re-oriented towards their corresponding clump-center hair vertices from their dry, combed position, where the default value for numberOfCVs is 3 (4 minus the root vertex), and the index for the current control vertex, i, ranges from 1-3.

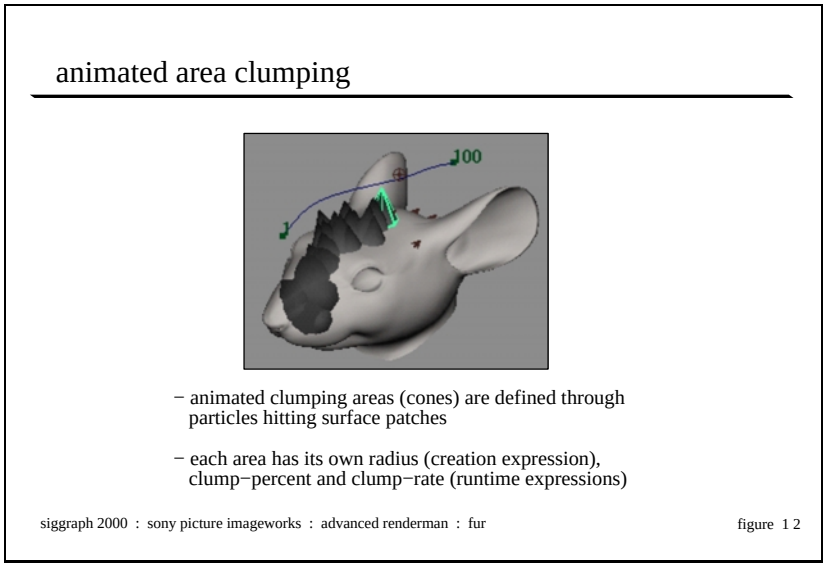
It is noted that both rate and percent operations change the lengths of clumps hairs. We have not, however, observed any visual artifacts even as these values are animated over the course of an animation.

static area clumping

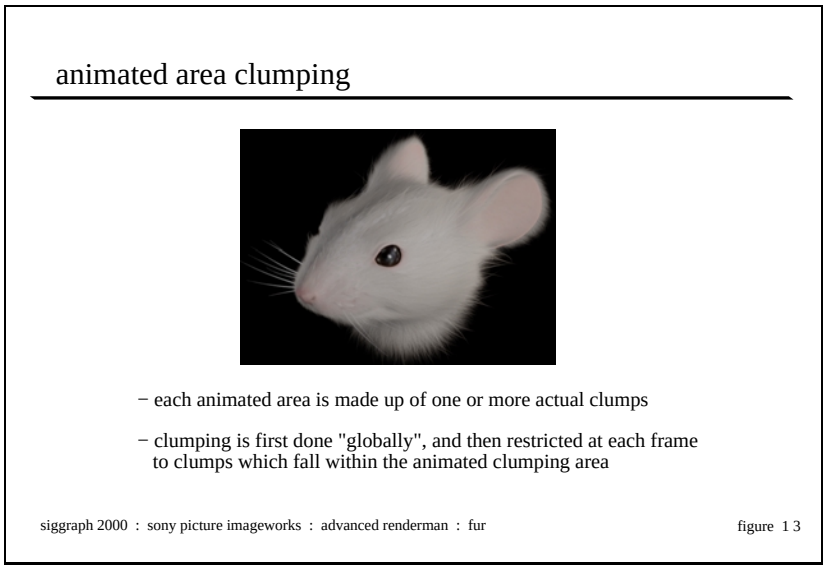
siggraph 2000 : sony picture imageworks : advanced renderman : fur figure 1 1

Image ©2000 by Columbia Pictures Industries Inc., All Rights Reserved

5.4.2 Animated Area Clumping

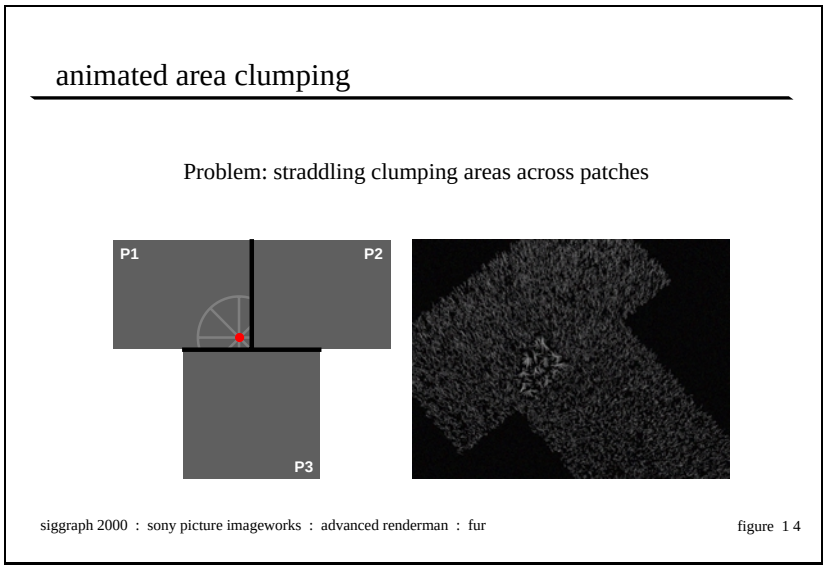


Animated area clumping is desirable if we want to simulate spouts of water or raindrops hitting the fur and making it increasingly wet. To simulate this, animated clumping areas are defined in an animation system through particles hitting surface patches. These particles originate from one or more emitters, whose attributes determine, for instance, the rate and spread of the particles. Once a particle hits a surface patch, a circular clumping area is created on the patch at that (u,v) location, with clump-percent, clump-rate and radius determined by a creation expression. Similar to static clump-size, this radius is converted into a corresponding u-radius and v-radius. Runtime expressions then define clump percent and rate, to determine how quickly and how much the fur “gets” wet. The above image shows a snapshot of animated clumping areas, where each area is visualized as a cone (the most recent cone is highlighted). In this case, one particle emitter is used and has been animated to follow the curve above the head.

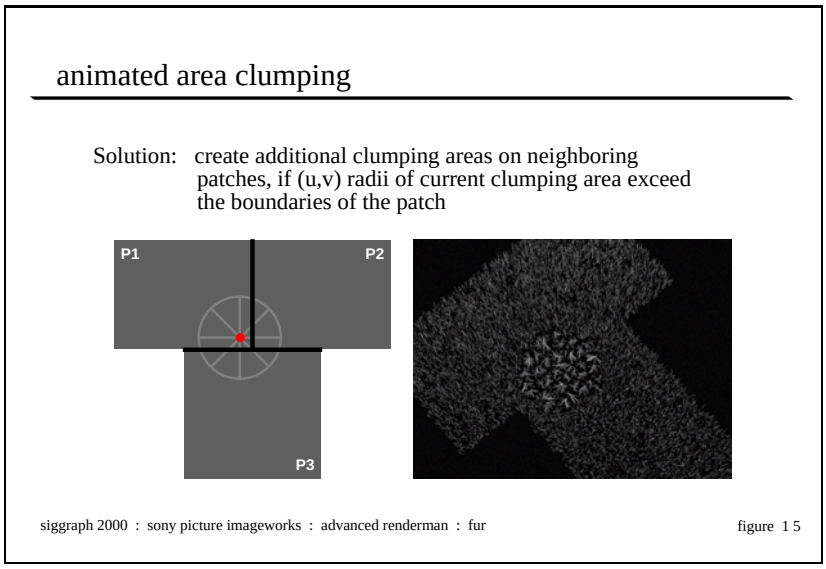


Once the animated clumping areas are defined for each frame of an animation, hairs which fall within these clumping areas are clumped. Clumping areas are usually much bigger than a clump, thus containing several individual clumps. In our implementation, we first apply static area clumping to the whole model as described above, and then restrict the clumping procedure to hairs within the animated clumping areas at each frame (i.e. to clumps which fall within any clumping area), so that final hairs of clumps outside clumping areas are normally produced as “dry” hairs.

To determine whether a clump falls within a clumping area, the clump method checks at each frame whether the (u,v) distance between the clump-center hair of the clump and the center of the clumping area is within the (u,v) radii of the clumping area. For clumps in overlapping clumping areas, the values for clump-percent and rate are added to generate wetter fur. The above image shows the resulting clumped hairs in the areas defined in the previous image.



One problem with animated clumping areas has not been addressed yet: they can straddle surface patch boundaries as shown in this illustration, which shows a top view of a clumping area (grey-colored cone): the center of this clumping area (black dot) is on patch P1, but the area straddles onto patch P2 and P3 (across a T-junction; seams are drawn as bold lines). Since our clumping areas are associated with the surface which contains the center of the clumping area, i.e. the position where the particle hit, portions of a clumping area straddling neighboring patches are ignored. This would lead to discontinuities in clumping of the final fur.



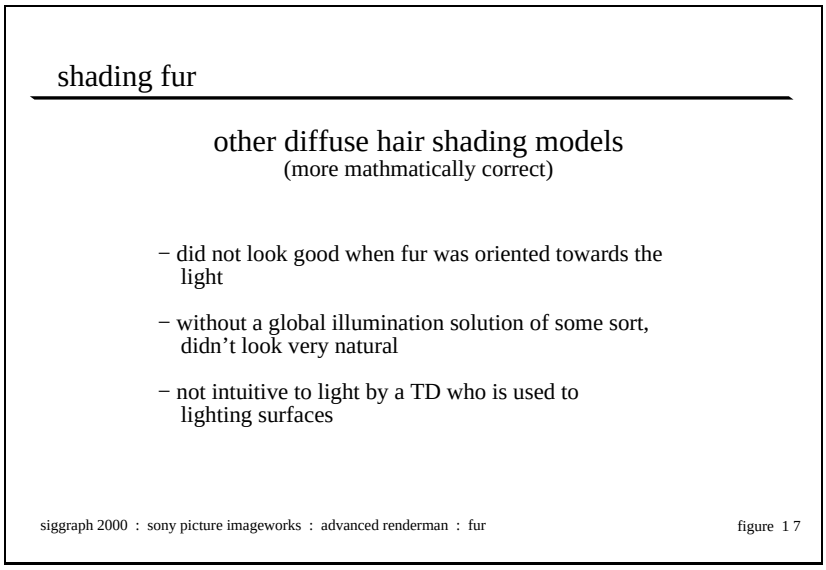
The solution presented here utilizes our surface seam and topology generator module: whenever

a new particle hits a surface, the clump method checks at each frame whether the (u,v) radii exceed the boundaries of that surface; if so, it calculates the (u,v) center and new (u,v) radii of this clumping area with respect to the affected neighboring patches. These "secondary" clumping areas on affected neighboring patches are then included in the calculations above to determine which clumps fall into clumping areas.

Image ©2000 by Columbia Pictures Industries Inc., All Rights Reserved

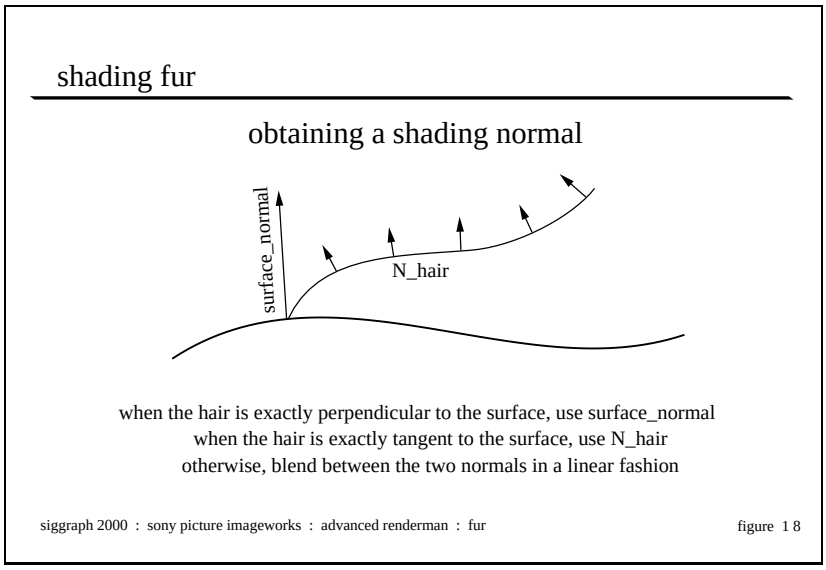
5.5 Shading

In order to render large amounts of hair quickly and efficiently, the geometric model of each hair is kept simple. As explained above, a hair is represented by a parametric curve with few control vertices (the default is four). We use the RenderMan `RiCurves` primitive to render the final hairs. This primitive consists of many small micro-polygons falling along a specified (hair) curve with their normals pointing directly towards the camera (like a 2-dimensional ribbon always oriented towards the camera). In comparison to long tubes, these primitives render extremely efficiently. However, we still need to solve the problem of how to properly shade these primitives as if they were thin hairs.



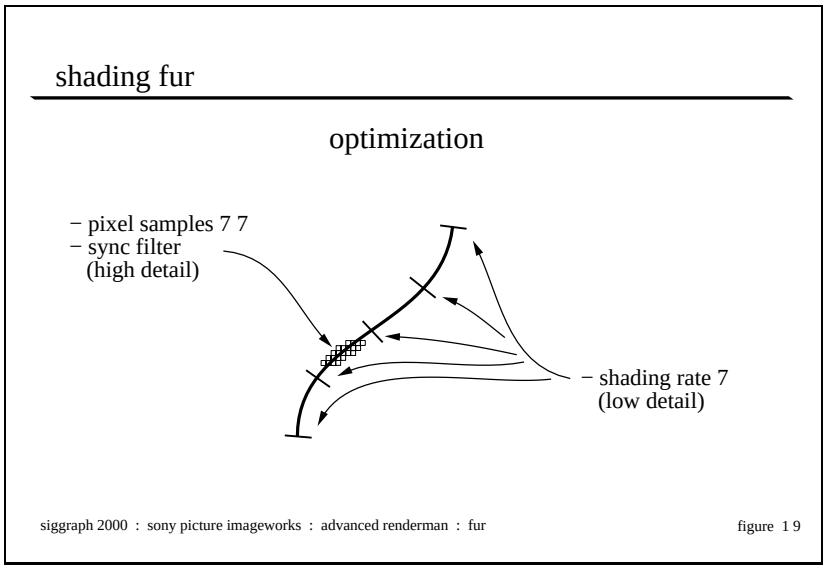
When using a more conventional lighting model for the fur (essentially a Lambert shading model applied to a very small cylinder) we came across a few complications. First, a diffuse model which integrates the contribution of the light as if the normal pointed in all directions around the cylinder leaves fur dark when oriented towards the light. This may be accurate but does not produce images that are very appealing. In our testing, we felt that the reason that this model didn't work as well was because a lot of the light that fur receives is a results of the bounce of the light off other fur in the character which could be modeled with some sort of global illumination solution, but would be very expensive.

The final (and for us the most significant) reason that we chose to develop a new diffuse lighting model for our fur was that it was not intuitive for a TD to light. Most lighting TD's have become very good at lighting surfaces and the rules by which those surfaces behave. So, our lighting model was designed to be lit in a similar way that you would light a surface while retaining variation over the length of the fur which is essential to get a high quality look.



In order to obtain a shading normal at the current point on the hair, we mix the surface normal vector at the base of the hair with the normal vector at the current point on the hair. The amount with which each of these vectors contributes to the mix is based on the angle between the tangent vector at the current point on the hair, and the surface normal vector at the base of the hair. The smaller this angle, the more the surface normal contributes to the shading normal. We then use a Lambertian model to calculate the intensity of the hair at that point using this shading normal. This has the benefit of allowing the user to light the underlying skin surface and then get very predictable results when fur is turned on. It also accounts for shading differences between individual hairs and along the length of each hair.

For the wet fur look, we change two aspects of our regular hair shading process. First, we increase the amount of specular on the fur. Second, we account for clumping in the shading model. Geometrically, as explained earlier, we model fur in clumps to simulate what actually happens when fur gets wet. In the shading model, for each hair, we calculate for each light, what side of the clump the hair is on with respect to the light's position, and then either darken or brighten the hair based on this. Thus, hairs on the side of a clump facing the light are brighter then hairs on a clump farther from the light.

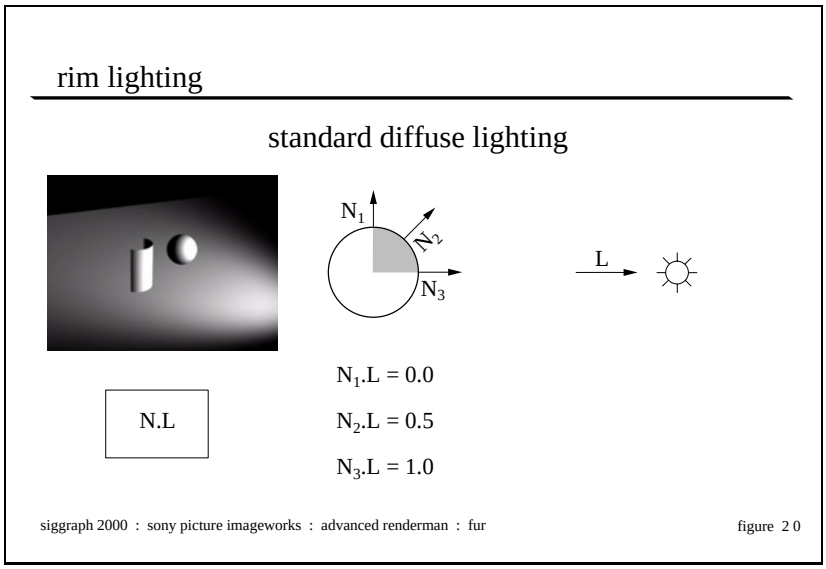


We learned a couple of things through experimentation about optimizing our fur renders. The level of detail requirements along the length of the fur (from one end to another) were very low. Our fur did not consist of many curves or sharp highlights so we could use a very high shading rate and simply allow RenderMan to interpolate the shaded values between the samples. This substantially reduced the number of times our shader code needed to be executed and sped up our renders.

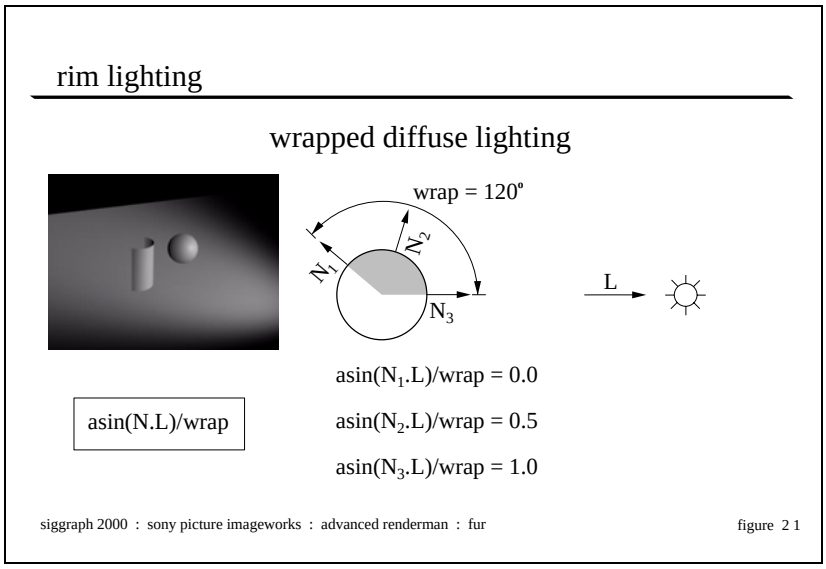
However, the value for the pixel samples needed to be quite high. We had lots of very thin overlapping curve primitives and without adequate pixel sampling they would alias badly. For most every fur render, our pixel samples needed to be 7 in both directions. We tried different filters as well and found visually that the sinc filter kept more sharpness and did not introduce any objectionable ringing.

5.6 Rim Lighting

To give Stuart his characteristic “hero” look it was necessary to have the capability of giving him a nice strong rim light that would wrap in a soft and natural way around the character.



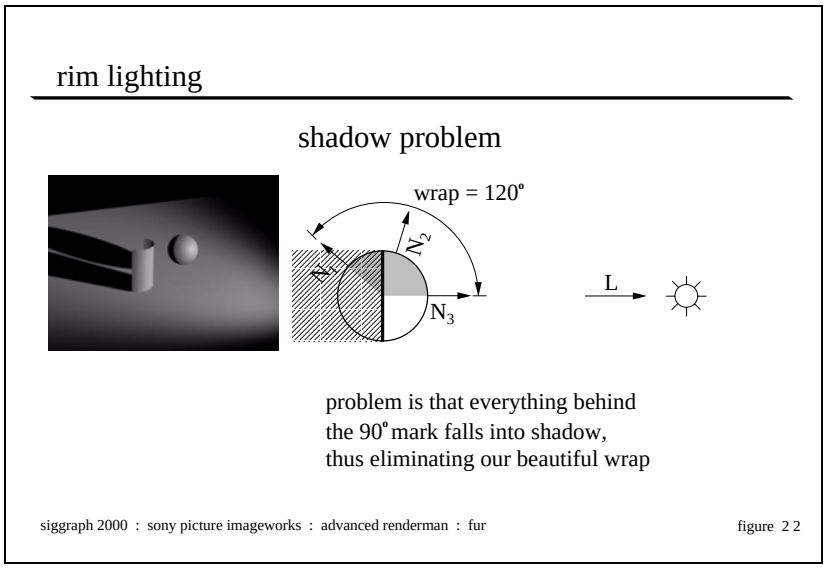
The previous slide illustrates a standard Lambertian diffuse shading model.



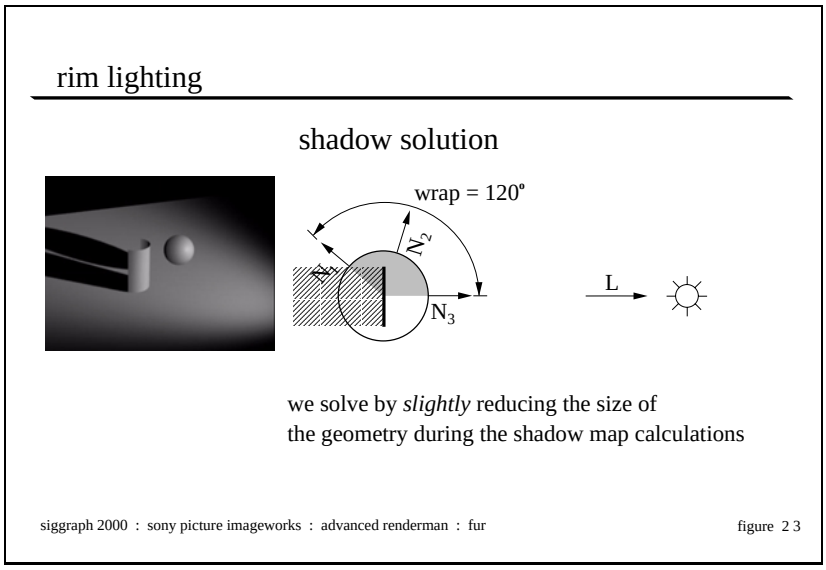
The first step in generating nicely wrapped lighting is to give the light the ability to reach beyond the 90 degree point on the objects. This has the effect of softening the effect of the light on the surface simulating an area light. The controls built into our lights had the ability to specify the end-wrapping-point in terms of degrees where a wrap of 90 degrees corresponded to the standard Lambertian shading model and higher values indicated more wrap. This control was natural for the lighting TD's to work with and gave predictable results.

For wrapped lights to calculate correctly, the thrid argument to the illuminance statement must be set to at least the same degree as the wrap value for the highest light. Otherwise lights will be culled out from the lighting calculations on the back of the surface and the light will not correctly wrap. In our implemention, the wrap parameter was set in each light and then passed into the shader (using message passing) where we used the customized diffuse calculation to take into account the light wrapping.

As an aside, wrapping the light "more" actually makes the light contribute more energy to the scene. For illustration purposes, the value of the light intensity was halved to generate the previous figure.



The first problem we encountered when putting these wrapped lights to practical use was with shadow maps. When you have shadow maps for an object, naturally the backside of the object will be made dark because of the shadow call. This makes it impossible to wrap light onto the backside of the object.



Fortunately RenderMan affords a few nice abilities to get around this problem. One way to solve this problem is by reducing the size of your geometry for the shadow map calculations. This way, the object will continue to cast a shadow (albeit a slightly smaller shadow) and the areas that need to catch the wrapped lighting are not occluded by the shadow map.

As a practical matter, shrinking an arbitrary object is not always an easy task. It can be done by displacing the object inwards during the shadow map calculations but this can be very expensive.

In the case of our hair, we wrote opacity controls into the hair shader that were used to drop the opacity of the hair to 0.0 a certain percentage of the way down their length. It is important to note that the surface shader for an object is respected during shadow map calculation and if you set the opacity of a shading point to 0.0, it will not register in the shadow map. The value which is considered “transparent” can be set with the following rib command:

```
Option "limits" "zthreshold" [0.5 0.5 0.5]
```

Lastly and most simply, you can blur the shadow map when making the `shadow()` call. If your requirements are for soft shadows anyway this is probably the best solution of all. It allows for the falloff of the light to “reach around” the object and if there is any occlusion caused by the shadow, it will be occluded softly which will be less objectionable.

For *Stuart Little*, we used a combination of shortening the hair for the shadow map calculations and blurring our shadows to be able to read the appropriate amount of wrap.

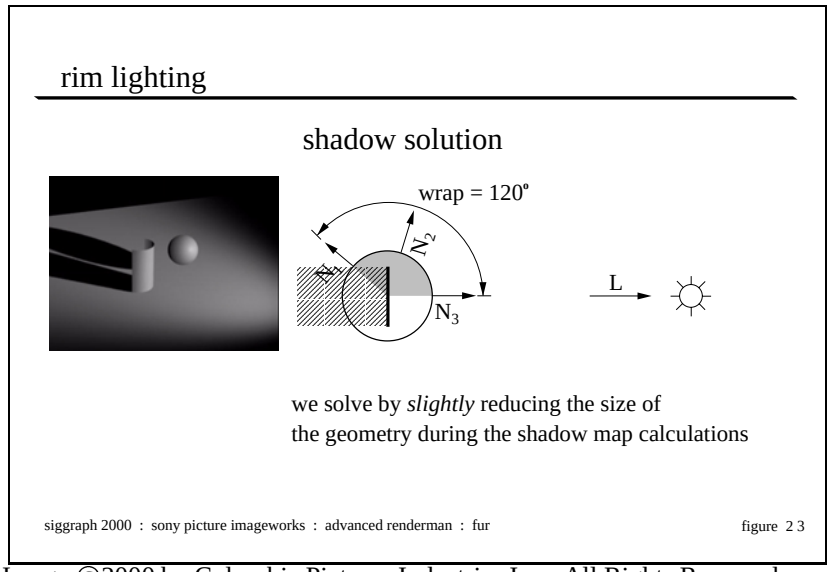


Image ©2000 by Columbia Pictures Industries Inc., All Rights Reserved

5.7 Other Tricks: Deformation System

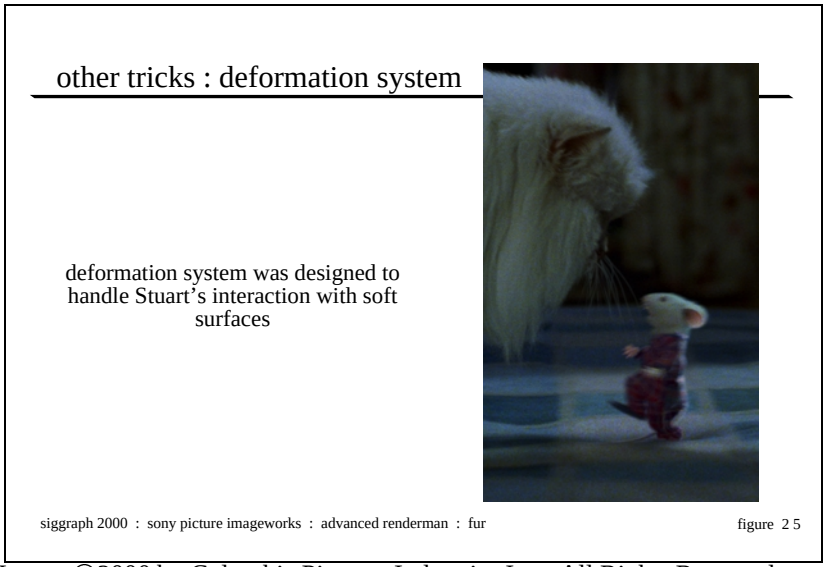
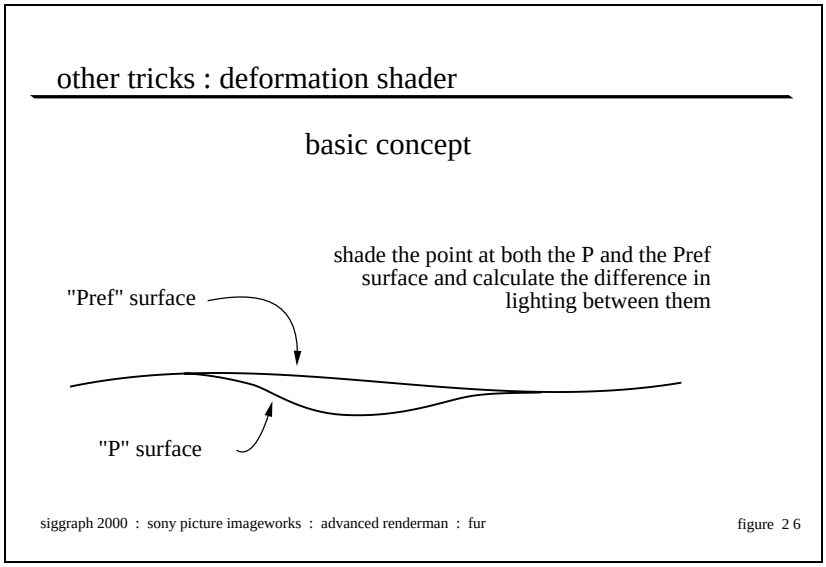


Image ©2000 by Columbia Pictures Industries Inc., All Rights Reserved

Stuart Little called for a number of shots where Stuart needed to interact with a soft environment. For these shots, we developed a system in RenderMan to create realistic animation of the surfaces

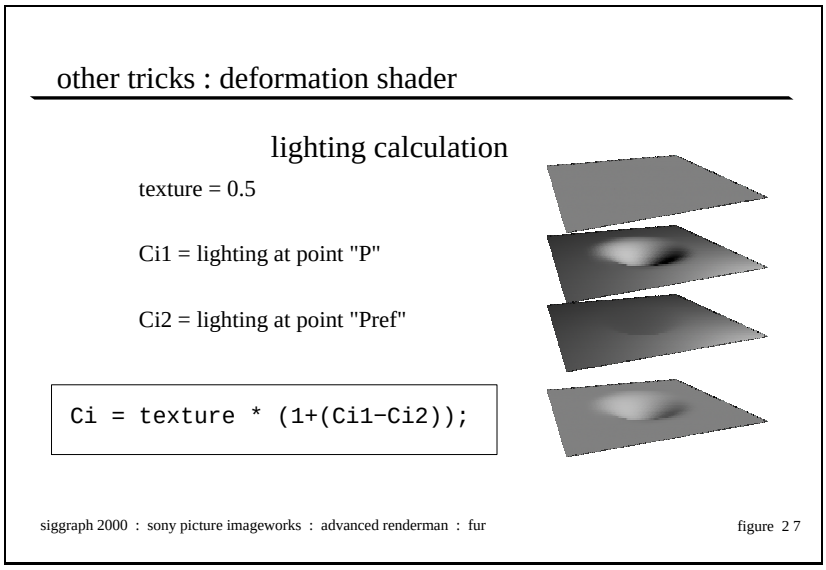
which were generally already in the plate photography. These surfaces needed to be manipulated to appear to be deforming under Stuart's feet and body.



The basic concept behind the system is to create two geometries. The first which represents the surface as it is represented in the plate. If the shot is moving or the surface was animating in the sequence, that surface may be matchmoved frame by frame. For purposes of the discussion of the shader we will refer to this as the "Pref" surface.

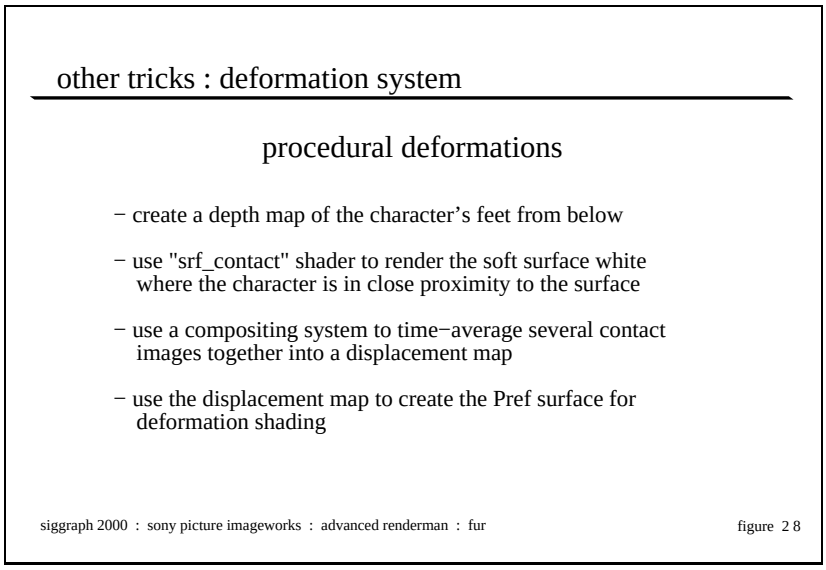
The second geometry represents the position of the surface when in contact with Stuart. We generate this surface in a variety of ways (one of which will be discussed later), but the basic idea is that the "P" surface is the deformed surface.

To "warp" the plate from the `Pref` to the `P` position is simply a matter of projecting the plate through the scene's camera onto the `Pref` geometry and then applying that texture to the `P` shading point (see the shader in the appendix for implemenation details).



The second thing that needs to be taken into account to make the deformation of the surface appear realistic is the lighting. The approach we take is to measure the lighting intensity on the `P` geometry, measure the lighting on the `Pref` geometry and use the equation illustrated in the previous figure to calculate the brightness change to the plate. This will correctly approximate what would have happened on the set had the same deformation taken place.

This approach generally worked fairly well as long as it wasn't taken to the extreme. The resolution of the input texture to the system was only the resolution of the film scans of the negative and there is some amount of softening that takes place while projecting and rendering the surface in RenderMan. Furthermore, if the stretching is very significant, issues such as the film grain of the plate and other details cannot be ignored. We also found it useful to add a set of color correction parameters that could control how the brightness changes to the plate were specifically applied so that the actual geometry deformation may not need to be re-animated to make a particular part of the deformation appear a little lighter or darker.



Creating the position of the deformed (or the `P`) object through entirely procedural methods is an interesting task which was developed in RenderMan. The first step is to create the "depth map" of the character's feet (or the part of the body that needs to do the interaction with the surface) from the bottom.

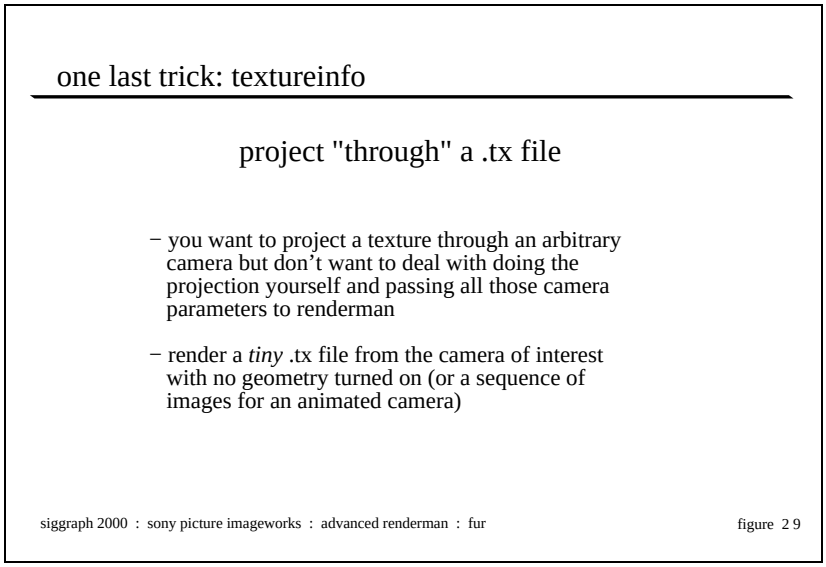
Next, use the depth map as a data file to drive a shader named `srf_contact.sl` which is included as an appendix to these notes. `srf_contact` is applied to the surface to be deformed and will turn to a 1.0 value when the character is within a specified distance from the surface and will gradually fade off as the character gets further away from the surface along the direction vector of the camera used to render the depth file. In our case, this process created a set of images with little white feet that would fade on and off as Stuart's feet got closer to and further from the surface. This element is named the "contact pass."

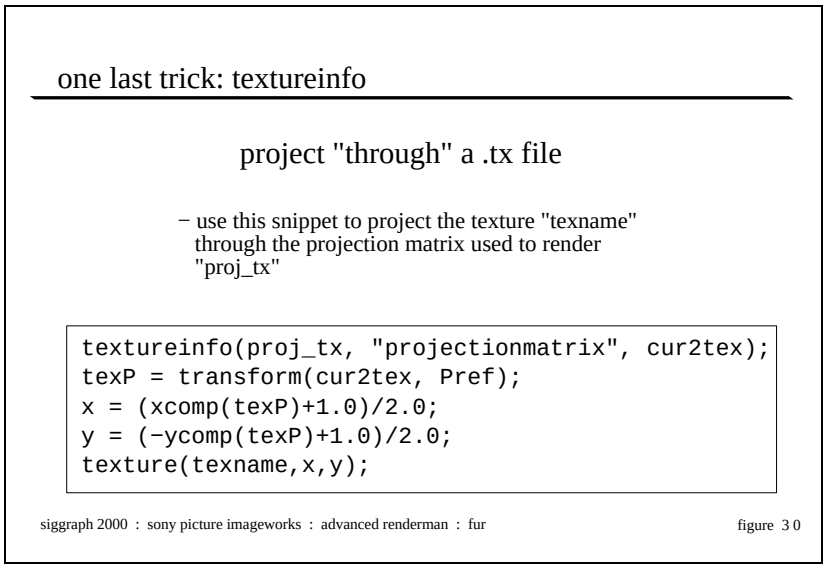
The next step in the process is to sample together several of these images into a new sequence. We used some time filtering techniques by which frame 10 of the sequence would consist of a weighted average of the last 6 frames of the "contact pass." This enabled a single footprint to leave an impression that would dissapear quickly but not instantaneously in the final deformations. The amount of blur applied to the images can also be animated over time generating footprints that soften as they decompress. This new sequence is generated as a series of "deformation textures."

The final step is to use those deformation textures as a displacement map to offset the `Pref` surface along the normal or another arbitrary vector to create the `P` surface. Naturally, you can do other interesting things with this displacement map such as add noise or other features to create interesting wrinkles to make the `P` surface appear more natural. But, once the two surfaces are created, the shader described above will calculate the difference between the two surfaces and should yield fairly natural looking results when used in combination with careful lighting.

5.8 Other Tricks: textureinfo

This is simply a handy RenderMan technique which has been available since the addition of the `textureinfo()` command in version 3.8 of *PRMan*. It is mentioned here because it has application in the procedural deformation system–when applying the “deformation textures” back onto the `Pref` geometry if you use this system you can simply project them “through” the “depth file” and they magically always end up in the right position, even with animated cameras. It also has other convenient uses including projecting multiple textures onto a single object for recreating surface detail from multiple still photographs.





5.9 Credits

The majority of the development of the fur technology for *Stuart Little* was performed by Clint Hanson and Armin Bruderlin. Armin also contributed in a large way to these course notes through his paper "A Method to Generate Wet and Broken-up Animal Fur" which was used nearly verbatim to describe the clumping techniques presented herein. Additionally, others who have contributed to the Sony Imageworks hair/fur pipeline are: Alan Davidson, Hiro Miyoshi, Bruce Navsky, Rob Engle, Jay Redd, Amit Agrawal as well as the Imageworks software, digital character, and production groups.

The author also wishes to acknowledge the support of Sony Imageworks for the publication of these notes. It should also be noted that some of the fur methods discussed in this course are techniques for which Sony Imageworks is currently applying for patents.

All figures and illustrations not indicated otherwise are Copyright 2000 by Sony Pictures Imageworks, all rights reserved.

5.10 Conclusion

There were several key concepts that made the fur pipeline for *Stuart Little* a workable solution. Delaying the creation of the largest portion of the dataset until render time was an efficient decision made early in the pipeline design that proved itself in terms of storage and processing time. For the clumping process, the idea of using both a realistic modeling technique combined with shading techniques yielded a strong "wet" fur look. And finally, the decision to use a lighting model which was based more on a TD's familiarity with lighting surfaces than what would be the most mathmatically accurate seemed to give more palatable results in less time, which improved the look of the film.

5.11 Shaders

5.11.1 Complete Fur Shader with Clumping and Specular Model

```
/* fur surface shader

   with clumping and specular model

   by Clint Hanson and Armin Bruderlin
*/

color
fnc_diffuselgt (color Cin;       /* Light Colour */
                point Lin;       /* Light Position */
                point Nin;       /* Surface Normal */
    )
{
    color Cout = Cin;
    vector LN, NN;
    float Atten;

    /* normalize the stuff */
    LN = normalize(vector(Lin));
    NN = normalize(vector(Nin));
```

```
    /* diffuse calculation */
    Atten = max(0.0,LN.NN);

    Cout *= Atten;

    return (Cout);
}


#define luminance(c) comp(c,0)*0.299 + comp(c,1)*0.587 + comp(c,2)*0.114


surface
srf_fur( /* Hair Shading... */
         float Ka   = 0.0287;
         float Kd   = 0.77;
         float Ks   = 1.285;
         float roughness1  = 0.008;
         float SPEC1  = 0.01;
         float roughness2  = 0.016;
         float SPEC2  = 0.003;
         float   start_spec = 0.3;
         float   end_spec = 0.95;
         float spec_size_fade  = 0.1;
         float illum_width  = 180;
         float var_fade_start = 0.005;
         float var_fade_end = 0.001;
         float clump_dark_strength = 0.0;

         /* Hair Color */
         color rootcolor  = color (.9714, .9714, .9714);
         color tipcolor = color (.519, .325, .125);
         color specularcolor  = (color(1) + tipcolor) / 2;
         color static_ambient  = color (0.057,0.057,0.057);

         /* Variables Passed from the rib... */
         uniform float hair_col_var  = 0.0;
         uniform float hair_length = 0.0;
         uniform normal surface_normal  = normal 1;
         varying vector clump_vect  = vector 0;
         uniform float hair_id   = 0.0; /* Watch Out... Across Patches */

    )
{
    vector T = normalize (dPdv); /* tangent along length of hair */
    vector V = -normalize(I);    /* V is the view vector */
    color Cspec = 0, Cdiff = 0;  /* collect specular & diffuse light */
    float Kspec = Ks;
    vector nL;
    varying normal nSN = normalize( surface_normal );
    vector S = nSN^T;       /* Cross product of the tangent along the hair
                               and surface normal */
    vector N_hair = (T^S);  /* N_hair is a normal for the hair oriented
                               "away" from the surface */
    vector norm_hair;
    float  l = clamp(nSN.T,0,1);  /* Dot of surface_normal and T, used
                                     for blending */
    float clump_darkening = 1.0;
    float T_Dot_nL = 0;
    float T_Dot_e = 0;
    float Alpha = 0;
```

```
float Beta = 0;
float Kajiya = 0;
float darkening = 1.0;
varying color final_c;

/* values from light */
uniform float nonspecular = 0;
uniform color SpecularColor = 1;

/* When the hair is exactly perpendicular to the surface, use the
   surface normal, when the hair is exactly tangent to the
   surface, use the hair normal Otherwise, blend between the two
   normals in a linear fashion
*/
norm_hair = (l * nSN) + ( (1-l) * N_hair);
norm_hair = normalize(norm_hair);

/* Make the specular only hit in certain parts of the hair--v is
   along the length of the hair
*/
Kspec *= min( smoothstep( start_spec, start_spec + spec_size_fade, v),
              1 - smoothstep( end_spec, end_spec - spec_size_fade, v ) );


/* Loop over lights, catch highlights as if this was a thin
   cylinder,

   Specular illumination model from:
     James T. Kajiya and Timothy L.  Kay (1989)
     "Rendering Fur with Three Dimensional Textures",
     Computer Graphics 23,3, 271-280
*/

illuminance (P, norm_hair, radians(illum_width)) {
    nL = normalize(L);

    T_Dot_nL = T.nL;
    T_Dot_e = T.V;
    Alpha = acos(T_Dot_nL);
    Beta = acos(T_Dot_e);

    Kajiya = T_Dot_nL * T_Dot_e + sin(Alpha) * sin(Beta);

    /* calculate diffuse component */
    if ( clump_dark_strength > 0.0 ) {
        clump_darkening =
               1 - ( clump_dark_strength *
                     abs(clamp(nL.normalize(-1*clump_vect), -1, 0)));
    } else {
        clump_darkening = 1.0;
    }

    /* get light source parameters */
    if ( lightsource("__nonspecular",nonspecular) == 0)
        nonspecular = 0;
    if ( lightsource("__SpecularColor",SpecularColor) == 0)
        SpecularColor = color 1;

    Cspec += (1-nonspecular) * SpecularColor * clump_darkening *
        ((SPEC1*Cl*pow(Kajiya, 1/roughness1)) +
         (SPEC2*Cl*pow(Kajiya, 1/roughness2)));
```

```
        Cdiff += clump_darkening * fnc_diffuselgt(Cl, L, norm_hair);
    }

    darkening = clamp(hair_col_var, 0, 1);

    darkening = 1 - (smoothstep( var_fade_end, var_fade_start,
                            abs(luminance(Kd*Cdiff))) * darkening);

    final_c = mix( rootcolor, tipcolor, v ) * darkening;

    Ci =  ((Ka*ambient() + Kd*Cdiff + static_ambient) * final_c
           + ((v) * Kspec * Cspec * specularcolor));

    Ci = clamp(Ci, color 0, color 1 );

    Oi = Os;
    Ci = Oi * Ci;
}
```

5.11.2 Deformation Shader

```
/*
   deformation surface shader

   projects a texture through the camera onto the Pref
   object and deforms it to the P position

   additionally, calculates the changes to shading on the surface
   measured by the change in diffuse lighting from the Pref to P.

   additional contrast and color controls are left as an exersize
   for the user.

   by Rob Bredow and Scott Stokdyk
*/

color
fnc_mydiffuse (color Cl;
               vector L;
               normal N)
{
    normal Nn;
    vector Ln;

    Nn = normalize(N);
    Ln = normalize(L);
    return Cl * max(Ln.Nn,0);
}


void
fnc_projectCurrentCamera(point P;
                         output float X, Y;)
{
    point Pndc = transform("NDC", P);

    X = xcomp(Pndc);
    Y = ycomp(Pndc);
}
```

```
surface
srf_deformation(
    string texname = "";  /* Texture to project */
    float debug = 0;      /* 0 = deformed lit image
                             1 = texture deformed with no lighting
                             2 = output lighting of the P object
                             3 = output lighting of the Pref object
                          */
    float Kd=1;           /* Surface Kd for lighting calculations */

    varying point Pref = point "shader" (0,0,0);
    )
{
    float x, y;
    color Ci0, Ci1, Ci2;
    normal N1, N2;
    float illum_width = 180;

    point Porig = Pref;

    fnc_projectCurrentCamera(Pref, x, y);

    if (texname != "") {
        Ci0 = texture(texname, x, y);
    }
    else Ci0 = 0.5;

    /* Calculate shading difference between P and Porig*/

    N = normalize(calculatenormal(P));
    N1 = faceforward(normalize(N),I);
    N = normalize(calculatenormal(Porig));
    N2 = faceforward(normalize(N),I);

    Ci1 = 0.0; Ci2 = 0.0;

    /* These lighting loops can be enhanced to calculate
       specular or reflection maps if needed */

    illuminance(P, N1, radians(illum_width)) {
        Ci1 += Kd * fnc_mydiffuse(Cl,L,N1);
    }

    illuminance(Porig, N2, radians(illum_width)) {
        Ci2 += Kd * fnc_mydiffuse(Cl,L,N2);
    }

    /* Difference in lighting acts as brightness control*/

    Ci = Ci0 * (1+(Ci1-Ci2));
    Oi = 1.0;

    if (debug == 1) {  /* output the texture - no lighting */
        Ci = Ci0;
    } else if (debug == 2) { /* output texture with P's lighting */
        Ci = Ci1;
    } else if (debug == 3) { /* output texture with Pref's lighting */
        Ci = Ci2;
    }
}
```

5.11.3 Contact ”Shadow” Shader

```
/*
**
** Render a contact shadow based on depth data derived from a light
** placed onto the surface which catches the contact shadow
**
** by Rob Engle and Jim Berney
*/

surface
srf_contact (
    string shadowname = ""; /* the name of the texture file */
    float samples = 10;     /* samples to take per Z lookup */
    float influence = 1.0;  /* world space distance in which
                                   effect is visible */
    float gamma = 0.5;      /* controls ramp on of effect over
                               distance */
    float maxdist = 10000;  /* how far is considered infinity */
    )
{
    /* get a matrix which transforms from current space to the
       camera space used when rendering the shadow map */
    uniform matrix matNl;
    textureinfo(shadowname, "viewingmatrix", matNl);

    /* get a matrix which transforms from current space to the
       screen space (-1..1) used when rendering the shadow map */
    uniform matrix matNP;
    textureinfo(shadowname, "projectionmatrix", matNP);

    /* transform the ground plane point into texture coordinates
       needed to look up the point in the shadow map */
    point screenP = transform(matNP, P);
    float ss = (xcomp(screenP) + 1) * 0.5;
    float tt = (1 - ycomp(screenP)) * 0.5;

    if (ss < 0 || ss > 1 || tt < 0 || tt > 1) {
        /* point being shaded is outside the region of the depth map */
        Ci = color(0);
    }
    else {
        /* get the distance from the shadow camera to the closest
           object as recorded in the shadow map */
        float mapdist = texture (shadowname, ss, tt, "samples", samples);

        /* transform the point on the ground plane into the shadow
           camera space in order to get the distance from the shadow
           camera to the ground plane */
        point cameraP = transform(matNl, P);

        /* the difference between the two distances is used
           to calculate the contact shadow effect */
        float distance = mapdist - zcomp(cameraP);

        distance = smoothstep(0, 1, distance/influence);
        distance = pow(distance, gamma);

        /* convert into a color (white=shadow) */
        Ci = (1.0-distance);
    }
}
```

Bibliography

Apodaca, A. A. and Gritz, L. (2000). *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. In George, J., editor, *Computer Graphics (SIGGRAPH '77 Proceedings)*, volume 11, pages 192–198.

Blinn, J. F. (1982). Light reflection functions for simulation of clouds and dusty surfaces. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16, pages 21–29.

Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546.

Cook, R. L., Carpenter, L., and Catmull, E. (1987). The Reyes image rendering architecture. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102.

Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 137–45. Monte Carlo distribution of rays to get gloss, translucency, penumbras, depth of field, motion blur.

Cook, R. L. and Torrance, K. E. (1981). A reflectance model for computer graphics. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, volume 15, pages 307–316.

Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Transactions on Graphics*, 1(1):7–24.

Glassner, A. e. (1989). *An Introduction to Ray Tracing*. Academic Press.

Glassner, A. S. (1995). *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers.

Goldman, D. B. (1997). Fake fur rendering. In Whitted, T., editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 127–134. ACM SIGGRAPH, Addison Wesley. ISBN 0-89791-896-7.

Gondek, J. S., Meyer, G. W., and Newman, J. G. (1994). Wavelength dependent reflectance functions. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 213–220. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

Greene, N. (1986a). Applications of world projections. In Green, M., editor, *Proceedings of Graphics Interface '86*, pages 108–114.

Greene, N. (1986b). Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29.

Hall, R. (1986). A characterization of illumination models and shading techniques. *The Visual Computer*, 2(5):268–77.

Hall, R. (1989). *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York. includes C code for radiosity algorithms.

Hanrahan, P. and Krueger, W. (1993). Reflection from layered surfaces due to subsurface scattering. In Kajiya, J. T., editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 165–174.

Hanrahan, P. and Lawson, J. (1990). A language for shading and lighting calculations. In Baskett, F., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 289–298.

He, X. D., Torrance, K. E., Sillion, F. X., and Greenberg, D. P. (1991). A comprehensive physical model for light reflection. In Sederberg, T. W., editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 175–186.

Kajiya, J. T. (1985). Anisotropic reflection models. In Barsky, B. A., editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 15–21.

Lafortune, E. P. F., Foo, S.-C., Torrance, K. E., and Greenberg, D. P. (1997). Non-linear approximation of reflectance functions. In Whitted, T., editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 117–126. ACM SIGGRAPH, Addison Wesley. ISBN 0-89791-896-7.

Nakamae, E., Kaneda, K., Okamoto, T., and Nishita, T. (1990). A lighting model aiming at drive simulators. In Baskett, F., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 395–404.

Oren, M. and Nayar, S. K. (1994). Generalization of lambert's reflectance model. In Glassner, A., editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 239–246. ACM SIGGRAPH, ACM Press. ISBN 0-89791-667-0.

Phong, B.-T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317.

Pixar (1989). *The RenderMan Interface, Version 3.1*. Pixar.

Poulin, P. and Fournier, A. (1990). A model for anisotropic reflection. In Baskett, F., editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 273–282.

Reeves, W. T., Salesin, D. H., and Cook, R. L. (1987). Rendering antialiased shadows with depth maps. In Stone, M. C., editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291.

Schlick, C. (1993). A customizable reflectance model for everyday rendering. In Cohen, M. F., Puech, C., and Sillion, F., editors, *Fourth Eurographics Workshop on Rendering*, pages 73–84. Eurographics. held in Paris, France, 14–16 June 1993.

Smits, B. E. and Meyer, G. M. (1989). Newton colors: Simulating interference phenomena in realistic image synthesis. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*.

Upstill, S. (1990). *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley.

Ward, G. J. (1992). Measuring and modeling anisotropic reflection. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 265–272.

Westin, S. H., Arvo, J. R., and Torrance, K. E. (1992). Predicting reflectance functions from complex surfaces. In Catmull, E. E., editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 255–264.

Whitted, T. and Cook, R. L. (1985). A comprehensive shading model. In *SIGGRAPH '85 Image Rendering Tricks seminar notes*.

Whitted, T. and Cook, R. L. (1988). A comprehensive shading model. In Joy, K. I., Grant, C. W., Max, N. L., and Hatfield, L., editors, *Tutorial: Computer Graphics: Image Synthesis*, pages 232–243. Computer Society Press.

Williams, L. (1978). Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274.